Using Rigging and Transfer to Animate 3D Characters

by

Ilya Baran

B.S., Massachusetts Institute of Technology (2003)
M.Eng., Massachusetts Institute of Technology (2004)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2010

Author .....
Department of Electrical Engineering and Computer Science
August 18, 2010

Certified by .....
Jovan Popović
Associate Professor
Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chairman, Department Committee on Graduate Theses

Using Rigging and Transfer to Animate 3D Characters

by

Ilya Baran

Submitted to the Department of Electrical Engineering and Computer Science
on August 18, 2010, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

Transferring a mesh or skeletal animation onto a new mesh currently requires significant manual effort. For skeletal animations, this involves rigging the character, by specifying how the skeleton is positioned relative to the character and how posing the skeleton drives the character's shape. Currently, artists typically manually position the skeleton joints and paint skinning weights onto the character to associate points on the character surface with bones. For this problem, we present a fully automatic rigging algorithm based on the geometry of the target mesh. Given a generic skeleton, the method computes both joint placement and the character surface attachment automatically. For mesh animations, current techniques are limited to transferring the motion literally using a correspondence between the characters' surfaces. Instead, I propose an example-based method that can transfer motion between far more different characters and that gives the user more control over how to adapt the motion to the new character.

Thesis Supervisor: Jovan Popović
Title: Associate Professor

Acknowledgments

The biggest thanks goes to my advisor, Jovan Popović. He was helpful and supportive from the moment I asked him to take me as a student and even after he was no longer at MIT. His guidance and flexibility made my graduate experience a pleasure.

The work in this thesis and outside was made possible by my collaborators, Daniel Vlasic, Wojciech Matusik, Eitan Grinspun, and Jaakko Lehtinen. Thanks to them, I learned to appreciate and enjoy the collaboration process.

A special thanks goes to Frédo Durand who often served as an informal second advisor, providing valuable feedback and being generally friendly. Sylvain Paris has also been extremely generous with writing help and sweets.

The MIT graphics group is a wonderful environment and I'd like to thank its members for their friendship. From lunches to teabreaks to squash to pre-SIGGRRAPH extreme hacking sessions, the graphics group is a great bunch. Although the friendships will last, I will definitely miss the day-to-day interaction.

I would also like to thank my friends outside the graphics group whom I could not see as often as I wanted to because of SIGGRAPH this or review that. There are too many to list, but I will mention Leo Goldmakher, Lev, Anya, and Anya Teytelman, Esya and Max Kalashnikov, Misha Anikin, Anya and Sasha Giladi, Lera and Dima Grinshpun, and Marina Zhurakhinskaya. Finally my thanks to my girlfriend Angelique, my sister Inna, and my parents Roman and Elena, for their unconditional support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Three-dimensional character animation is an essential component of modern computer games and motion pictures, both animated and live-action. As with any art form, a huge range of quality is possible, from crude amateur creations to completely believable digital characters like Gollum from The Lord of the Rings or Aslan from the Chronicles of Narnia. The result quality depends on many factors, such as the skill of the artist(s), the effort spent, whether the animation is static and precomputed or needs to be responsive and real-time. However, even low-quality results require a significant effort, much of which is repetitive, rather than creative. Automating repetitive tasks is a job for software tools, but in spite of recent progress in this area, producing character animation remains difficult for professionals and almost hopeless for amateurs. The goal of this thesis is to explore ways to automate and simplify the repetitive tasks, reducing the amount of effort that it takes to create an animated character.

1.1 Character Animation Pipeline

We start with a brief overview of how animated characters are currently created. First a character's geometry is constructed (*modeling*), then the character's degrees of freedom are specified (*rigging*), and finally a motion is applied to these degrees of freedom (*animation*).

1.1.1 Modeling

Most of the time character geometry is specified either as a polygon mesh, a set of splines, or a subdivision surface, which is eventually converted to a polygon mesh. While other representations are possible, such as implicit, volumetric, or point-based, working with them is more difficult and hence they are rarely used. The typical modeling workflow is to manually build a coarse mesh, apply subdivision to obtain a finer tessellation, and then use local operations that move many vertices simultaneously ("sculpting") to provide the detail. If the mesh needs to have a low polygon count, such as for video games, it is downsampled and the eliminated detail is placed into a normal map or a displacement map.

In production, modeling is done with in-house tools or large packages such as Maya. Sculpting is often done in specialized software, such as ZBrush or Mudbox. For amateur use and even for the professional artists, these tools are very difficult to learn and use, leading to a lot of effort to develop simpler modeling techniques. Existing mainstream tools require the user to be constantly aware of the surface representation: even though the user intends to create a smooth surface, he or she is forced to work with individual vertices, edges, and polygons. A common theme in modeling research is to try to hide the representation from the user. Welch and Witkin have done early work in this direction using triangle meshes that adjust themselves automatically to minimize energy [71, 72]. More recently, a variety of modeling prototypes have been proposed that allow a user to create 3D models from scratch with simple sketch-based controls [27, 26, 55, 54]. Another attempt at simplifying modeling is Cosmic Blobs, developed at SolidWorks, in part by the author. It allows the user to start with a smooth primitive surface and apply a sequence of deformations to create a shape. Because of limited scalability these tools have not made it into production and intuitive 3D modeling remains an unsolved problem.

1.1.2 Rigging

Once a character has been constructed, it usually has too many degrees of freedom for an animator to be able to control the character directly. For example, a mesh with as few as $5,000$ vertices has $15,000$ degrees of freedom—a huge number of controls. To make animation convenient, the artist must specify the meaningful degrees of freedom of the character. This observation was made explicitly in the context of vector art (rather than 3D meshes) by Ngo et al.: most configurations of a model are meaningless and constraints are required to produce interesting results [50]. Specifying the degrees of freedom that an artist will want to control in an animated character is called rigging.

The degrees of freedom of a rigged character are related to the desired animation quality. At the high end, rigs have hundreds of degrees of freedom and are specified as arbitrary deformations, combining physical simulation of the underlying musculoskeletal structure, mesh deformation, and general procedural techniques. A high-end rig can produce effects such as bulging muscles, folding skin, rippling fur, and skin sliding over bones and joints. Such rigs can take a team of artists months to construct. Clothing may be part of the rig, or it may be simulated separately. On the low end, a skilled artist can build a coarse rig with 20–30 degrees of freedom that allows skeletal motion in several hours. Most rigging is done at the low end: the cost of artists and performance requirements confine complicated custom rigs to high-budget films.

A common assumption for low-end body (as opposed to face or hair) rigging is that the degrees of freedom are essentially skeletal: that the character is attached to an underlying skeleton and moving the bones drives the character. The skeleton is non-physical: bones are be specified as line segments and a pose is specified by applying a rigid motions to each bone. To build a skeletal rig, the artist constructs the skeleton, specifying the degrees of freedom of the bones, positions the bones correctly relative to the character, and specifies how the character's surface deforms for a given pose of the skeleton.

The process of attaching the character to the skeleton is called skinning and several

Figure 1-1: Large joint deformations of an arm (left) lead to joint collapsing artifacts.

general techniques are used. The simplest is to attach each mesh vertex to a single bone: if a rigid motion $\mathbf{T}_b$ is applied to bone b, then that rigid motion is also applied to all vertices attached to bone b. This method produces obvious and ugly artifacts when two adjacent vertices are attached to different bones, so it is almost never used.

A simple way to avoid this problem is to apply the transformations of multiple bones to a vertex and take a linear combination of the results: each vertex $\mathbf{v}$ has a weight w_b for each bone, indicating how much it is attached to that bone, and the posed position of the vertex is $\sum_b w_b \mathbf{T}_b(\mathbf{v})$. Because the weights are expressing a combination of the transformations, they are nonnegative and $\sum_b w_b = 1$. By smoothly varying the weights w across the character surface, it is possible to obtain smooth deformations for poses. This method is known by many names: skeletal subspace deformation, matrix palette skinning, or linear blend skinning (LBS). From an artist's point of view, using LBS means specifying a w for each mesh vertex for each bone. This is a lot of data and it is not always intuitive to specify. Production software typically allows the artist to specify the weights by "painting" them onto the surface, but this is labor intensive and somewhat of a black art.

Many attempts at simplifying rigging have focused on automatically extracting a "curve-skeleton" from a character mesh (or volume). Curve-skeletons are not rigorously defined, but a curve-skeleton is understood to be a complex of curves inside the shape that captures the important geometric features. Curve-skeleton extraction methods have been developed based on Voronoi diagrams [61], distance fields [5], volumetric thinning [51, 6], Reeb graphs [63], potential fields [44], or clustering [35]. A recent survey [12] describes and classifies many of the various methods. A curve-skeleton can be used for skeletal animation and some algorithms for extracting a

curve-skeleton also compute a surface attachment [61, 35, 67]. While an extracted curve-skeleton is a viable automatic rigging tool, it is hard to control its complexity for characters with complex geometry. It also carries no semantic information, such as which part of the extracted skeleton is an arm, and an existing animation therefore cannot be directly applied.

While linear blend skinning avoids the obvious discontinuities, it still suffers from well-known problems. The fundamental issue is that linearly interpolating between a 0 degree rotation and a 180 degree rotation yields a singular matrix, rather than a 90 degree rotation. In practice, this leads to joint collapsing effects, as shown in Figure 1-1. Artists get around this problem (as well as LBS's inability to capture muscle bulges and other nonskeletal effects) by introducing more bones or using corrections for certain poses. At the same time, there has been research work that aims to address the drawbacks of linear blend skinning without increasing complexity. Some recent work has attempted to improve on the quality of linear blend skinning, without sacrificing real-time performance [37, 69, 36]. Within the same framework as LBS, i.e., mesh vertices attached to multiple bones with weights, the best current method is dual quaternion blending [36], which avoids joint collapsing by combining transformations in a way that preserves rigid motions.

However even without collapsing artifacts, the expressiveness of LBS is limited. To capture non-skeletal effects, such as a bulging biceps, folding cloth, or skin sliding over bone, several example-based methods have been developed [41, 40, 68]. The drawback of such methods is, of course, that they require multiple example poses of the character, which require artist effort to provide.

Recently, an alternative rigging method has become popular. Rather than embedding a skeleton into the character, the character is itself embedded in a coarse polyhedral cage. Each point of the character is expressed as a linear combination of the cage vertices and deforming the cage results in the deformation of the character [17, 33, 32, 42]. Such methods have comparable deformation performance to linear blend skinning, while avoiding joint collapses and allowing a greater freedom in representing deformation. However, cages require careful construction and are more

difficult to control than skeletons.

A barrier to realism in the methods discussed above is that they are purely static: the deformed surface at each point in time is a function of the control at that point in time. In reality, however, the surface of a moving character is subject to temporal effects like inertia. Animations that properly incorporate these effects tend to look livelier and less stiff. There have been attempts to incorporate such physical effects into animation rigs [10], but they reduce performance to levels unsuitable for computer games, for example.

1.1.3 Animation

Controlling the degrees of freedom of a character poses another challenge. The two most common approaches to this problem are motion capture and keyframing. A wide variety of motion capture setups exist. In the majority of setups, an actor attaches "markers" to different parts of his or her body and the motion capture system tracks these markers and reconstructs the actor's skeletal motion. The markers may be active or passive, optical, acoustic, inertial or magnetic. The captured motion can drive the skeleton of a rigged model. Recently, a number of methods have been proposed to capture the surface of an actor, not just his or her skeletal pose [57, 15, 66, 14]. This allows capturing non-skeletal effects and even garment motion. These methods use multiple synchronized video cameras to record a performance and process the video to obtain a shape for each frame. In general, motion capture systems are expensive, and often require a tedious clean-up step, but they can reproduce the actor's motion with millimeter (or even better) precision.

While the primary benefit of motion capture is that it reproduces the motion of the actor exactly, this can also be a drawback. Sometimes the virtual character needs to move differently from how a human can move. When motion capture is available and the desired motion is not far from what can be captured, editing tools exist for both skeletal [21] and mesh [39] animation. In other cases, however, motion needs to be created from scratch and keyframing is used. Keyframing involves specifying the relevant degrees of freedom explicitly at certain points in time and interpolating to

get a smooth trajectory. Keyframing provides maximum flexibility, but is very labor intensive and requires a great deal of artistic skill. As a result, several alternatives have been proposed, mainly for simple models and motions. Sketching [62] or stylized acting [16] provide simple input methods. Placing keyframes in a space other than the timeline leads to spatial keyframing [29] or (in the case of 2D characters) configuration modeling [50].

In some cases, it is possible to embed the character in a simulated physical environment and either use a controller to drive the character's skeleton [65, 52, 23, 74, 13], or use spacetime optimization to find a motion that optimizes some objective (such as minimizing energy expenditure), while respecting physical laws [73, 49]. These techniques are rarely used in practice because constructing robust controllers that produce believable motion is very difficult and spacetime optimization is very high-dimensional and therefore slow and unstable.

To obtain more sophisticated animations with less work than keyframing or physical simulation, several methods for reusing existing motion have been proposed. For skeletal motion, methods exist for retargetting to new characters [20], changing the style [25], mapping to completely new motion [24], or learning a probabilistic model that can synthesize new motion automatically [8].

For mesh animation, as opposed to skeletal animation, there has been less work on reusing motion. The only method developed to date specifically for this purpose is deformation transfer [58]. It uses a correspondence between the surfaces of two characters to transfer the motion of one character onto another. However, the characters on which this works need to be similar, providing the surface correspondence may be difficult, and the motion is transferred without modification. Our work aims to address these drawbacks.

1.2 Contributions

This thesis presents two algorithms designed to simplify character animation. The first algorithm, automatic rigging, enables skeletal rigging to be done fully automati-

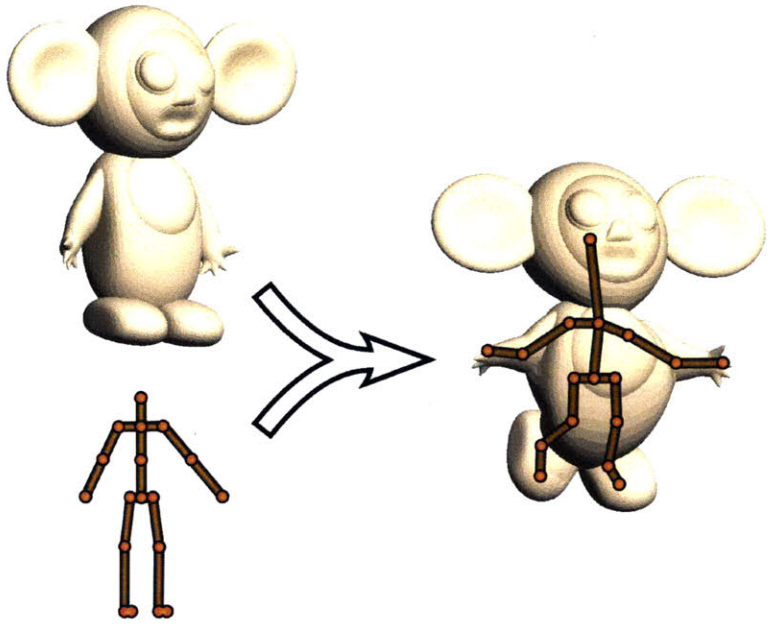

Figure 1-2: The automatic rigging method presented in this thesis allowed us to implement an easy-to-use animation system, which we called Pinocchio. In this example, the triangle mesh of a jolly cartoon character is brought to life by embedding a skeleton inside it and applying a walking motion to the initially static shape.

cally, allowing a user to apply an existing skeletal animation to a new character. The second algorithm, semantic deformation transfer, expands the class of mesh animations that is possible to obtain from existing mesh animations.

1.2.1 Automatic Rigging

While recent modeling techniques have made the creation of 3D characters accessible to novices and children, bringing these static shapes to life is still not easy. In a conventional skeletal animation package, the user must rig the character manually. This requires placing the skeleton joints inside the character and specifying which parts of the surface are attached to which bone. The tedium of this process makes simple character animation difficult.

We present a system that eliminates this tedium to make animation more accessible for children, educators, researchers, and other non-expert animators. A design

goal is for a child to be able to model a unicorn, click the "Quadruped Gallop" button, and watch the unicorn start galloping. To support this functionality, we need a method (as shown in Figure 1-2) that takes a character, a skeleton, and a motion of that skeleton as input, and outputs the moving character.

1.2.2 Semantic Deformation Transfer

When a skeletal model cannot adequately capture the desired complexity of the deformation, reusing existing mesh animations to drive new ones is an attractive alternative. Existing methods apply when a surface correspondence is defined between the characters and the characters move in the same manner. While existing techniques can map a horse motion to a camel motion, they cannot, for example, transfer motion from a human to a flamingo, because the knees bend in a different direction. To support transferring mesh animation to very different characters, we developed semantic deformation transfer, which preserves the semantics of the motion, while adapting it to the target character. The user provides several pairs of example poses that exercise the relevant degrees of freedom to specify semantic correspondence between two characters. For example, to transform a human walk onto a flamingo, the user needs to provide poses of the human with bent knees and corresponding poses of the flamingo with knees bent backwards. The algorithm then learns a mapping between the two characters that could then be applied to new motions. Semantic deformation transfer can also be seen as a generalization of keyframing, in which keyframes are placed in the source character's pose space.

Chapter 2

Automatic Rigging

2.1 Overview

Our automatic rigging algorithm consists of two main steps: skeleton embedding and skin attachment. Skeleton embedding computes the joint positions of the skeleton inside the character by minimizing a penalty function. To make the optimization problem computationally feasible, we first embed the skeleton into a discretization of the character's interior and then refine this embedding using continuous optimization. The skin attachment is computed by assigning bone weights based on the proximity of the embedded bones smoothed by a diffusion equilibrium equation over the character's surface.

Our design decisions relied on three criteria, which we also used to evaluate our system:

- **Generality:** A single skeleton is applicable to a wide variety of characters: for example, our method can use a generic biped skeleton to rig an anatomically correct human model, an anthropomorphic robot, and even something that has very little resemblance to a human.
- **Quality:** The resulting animation quality is comparable to that of modern video games.
- **Performance:** The automatic rigging usually takes under one minute on an

everyday PC.

A key design challenge is constructing a penalty function that penalizes undesirable embeddings and generalizes well to new characters. For this, we designed a maximum-margin supervised learning method to combine a set of hand-constructed penalty functions. To ensure an honest evaluation and avoid overfitting, we tested our algorithm on 16 characters that we did not see or use during development. Our algorithm computed a good rig for all but 3 of these characters. For each of the remaining cases, one joint placement hint corrected the problem.

We simplify the problem by making the following assumptions. The character mesh must be the boundary of a connected volume. The character must be given in approximately the same orientation and pose as the skeleton. Lastly, the character must be proportioned roughly like the given skeleton.

We introduce several new techniques to solve the automatic rigging problem:

- A maximum-margin method for learning the weights of a linear combination of penalty functions based on examples, as an alternative to hand-tuning (Section 2.3.3).

- An A^*-like heuristic to accelerate the search for an optimal skeleton embedding over an exponential search space (Section 2.3.4).

- The use of Laplace's diffusion equation to generate weights for attaching mesh vertices to the skeleton using linear blend skinning (Section 2.4). This method can also be useful in existing 3D packages when the skeleton is manually embedded.

Our prototype system, called Pinocchio, rigs the given character using our algorithm. It then transfers a motion to the character using online motion retargetting [11] to eliminate footskate by constraining the feet trajectories of the character to the feet trajectories of the given motion.

2.2 Related Work

Character Animation Most prior research in character animation, especially in 3D, has focused on professional animators; very little work is targeted at novice users. Recent exceptions include Motion Doodles [62] as well as the work of Igarashi et al. on spatial keyframing [29] and as-rigid-as-possible shape manipulation [28]. These approaches focus on simplifying animation control, rather than simplifying the definition of the articulation of the character. In particular, a spatial keyframing system expects an articulated character as input, and as-rigid-as-possible shape manipulation, besides being 2D, relies on the constraints to provide articulation information. The Motion Doodles system has the ability to infer the articulation of a 2D character, but their approach relies on very strong assumptions about how the character is presented.

Some previous techniques for character animation infer articulation from multiple example meshes [40]. Given sample meshes, mesh-based inverse kinematics [59] constructs a nonlinear model of meaningful mesh configurations, while James and Twigg [30] construct a linear blend skinning model, albeit without a meaningful hierarchical skeleton. However, such techniques are unsuitable for our problem because we only have a single mesh. We obtain the character articulation from the mesh by using the given skeleton not just as an animation control structure, but also as an encoding of the likely modes of deformation.

Skeleton Extraction Although most skeleton-based prior work on automatic rigging focused on skeleton extraction, for our problem, we advocate skeleton embedding. A few approaches to the skeleton extraction problem are representative. Teichmann and Teller [61] extract a skeleton by simplifying the Voronoi skeleton with a small amount of user assistance. Liu et al. [44] use repulsive force fields to find a skeleton. In their paper, Katz and Tal [35] describe a surface partitioning algorithm and suggest skeleton extraction as an application. The technique in Wade [67] is most similar to our own: like us, they approximate the medial surface by finding discontinuities in the distance field, but they use it to construct a skeleton tree.

For the purpose of automatically animating a character, however, skeleton embedding is much more suitable than extraction. For example, the user may have motion data for a quadruped skeleton, but extracting the skeleton from the geometry of a complicated quadruped character, is likely to lead to an incompatible skeleton topology. The anatomically appropriate skeleton generation by Wade [67] ameliorates this problem by techniques such as identifying appendages and fitting appendage templates, but the overall topology of the resulting skeleton may still vary. For example, for the character in Figure 1-2, ears may be mistaken for arms. Another advantage of embedding over extraction is that the given skeleton provides information about the expected structure of the character, which may be difficult to obtain from just the geometry. So although we could use an existing skeleton extraction algorithm and embed our skeleton into the extracted one, the results would likely be undesirable. For example, the legs of the character in Figure 1-2 would be too short if a skeleton extraction algorithm were used.

Template Fitting Animating user-provided data by fitting a template has been successful in cases when the model is fairly similar to the template. Most of the work has been focused on human models, making use of human anatomy specifics, e.g. [47]. For segmenting and animating simple 3D models of characters and inanimate objects, Anderson et al. [1] fit voxel-based volumetric templates to the data.

Skinning Almost any system for mesh deformation (whether surface based [43, 75] or volume based [76]) can be adapted for skeleton-based deformation. Teichmann and Teller [61] propose a spring-based method. Unfortunately, at present, these methods are unsuitable for real-time animation of even moderate size meshes. Because of its simplicity and efficiency (and simple GPU implementation), and despite its quality shortcomings, linear blend skinning (LBS), also known as skeleton subspace deformation, remains the most popular method used in practice.

Most real-time skinning work, e.g. [40, 68], has focused on improving on LBS by inferring the character articulation from multiple example meshes. However, such

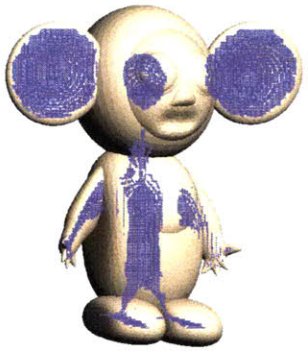

Figure 2-1: Approximate Medial Surface

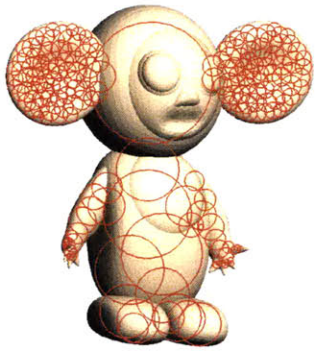

Figure 2-2: Packed Spheres

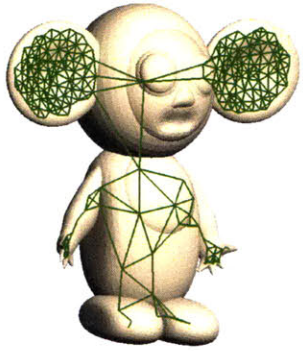

Figure 2-3: Constructed Graph

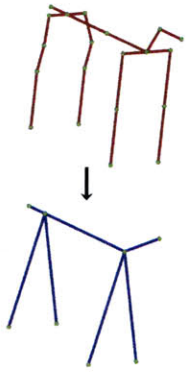

Figure 2-4: The original and reduced quadruped skeleton

techniques are unsuitable for our problem because we only have a single mesh. Instead, we must infer articulation by using the given skeleton as an encoding of the likely modes of deformation, not just as an animation control structure.

To our knowledge, the problem of finding bone weights for LBS from a single mesh and a skeleton has not been sufficiently addressed in the literature. Previous methods are either mesh resolution dependent [35] or the weights do not vary smoothly along the surface [67], causing artifacts on high-resolution meshes. Some commercial packages use proprietary methods to assign default weights. For example, Autodesk Maya 7 assigns weights based solely on the vertex proximity to the bone, ignoring the mesh structure, which results in serious artifacts when the mesh intersects the Voronoi diagram faces between logically distant bones.

2.3 Skeleton Embedding

Skeleton embedding resizes and positions the given skeleton to fit inside the character. This can be formulated as an optimization problem: "compute the joint positions such that the resulting skeleton fits inside the character as nicely as possible and looks like the given skeleton as much as possible." For a skeleton with s joints (by "joints," we mean vertices of the skeleton tree, including leaves), this is a $3s$-dimensional problem with a complicated objective function. Solving such a problem directly using continuous optimization is infeasible.

Pinocchio therefore discretizes the problem by constructing a graph whose vertices represent potential joint positions and whose edges are potential bone segments. This is challenging because the graph must have few vertices and edges, and yet capture all potential bone paths within the character. The graph is constructed by packing spheres centered on the approximate medial surface into the character and by connecting sphere centers with graph edges. Pinocchio then finds the optimal embedding of the skeleton into this graph with respect to a discrete penalty function. It uses the discrete solution as a starting point for continuous optimization.

To help with optimization, the given skeleton can have a little extra information in the form of joint attributes: for example, joints that should be approximately symmetric should be marked as such; also some joints can be marked as "feet," indicating that they should be placed near the bottom of the character. We describe the attributes Pinocchio uses in Appendix A.1. These attributes are specific to the skeleton but are independent of the character shape and do not reduce the generality of the skeletons.

2.3.1 Discretization

Before any other computation, Pinocchio rescales the character to fit inside an axis-aligned unit cube. As a result, all of the tolerances are relative to the size of the character.

Distance Field To approximate the medial surface and to facilitate other computations, Pinocchio computes a trilinearly interpolated adaptively sampled signed distance field on an octree [18]. It constructs a kd-tree to evaluate the exact signed distance to the surface from an arbitrary point. It then constructs the distance field from the top down, starting with a single octree cell and splitting a cell until the exact distance is within a tolerance τ of the interpolated distance. We found that $\tau = 0.003$ provides a good compromise between accuracy and efficiency for our purposes. Because only negative distances (i.e. from points inside the character) are important, Pinocchio does not split cells that are guaranteed not to intersect the character's interior.

Approximate Medial Surface Pinocchio uses the adaptive distance field to compute a sample of points approximately on the medial surface (Figure 2-1). The medial surface is the set of C^1-discontinuities of the distance field. Within a single cell of our octree, the interpolated distance field is guaranteed to be C^1, so it is necessary to look at only the cell boundaries. Pinocchio therefore traverses the octree and for each cell, looks at a grid (of spacing τ) of points on each face of the cell. It then computes the gradient vectors for the cells adjacent to each grid point—if the angle between two of them is 120° or greater, it adds the point to the medial surface sample. We impose the 120° condition because we do not want the "noisy" parts of the medial surface—we want the points where skeleton joints are likely to lie. For the same reason, Pinocchio filters out the sampled points that are too close to the character surface (within 2τ). Wade discusses a similar condition in Chapter 4 of his thesis [67].

Sphere Packing To pick out the graph vertices from the medial surface, Pinocchio packs spheres into the character as follows: it sorts the medial surface points by their distance to the surface (those that are farthest from the surface are first). Then it processes these points in order and if a point is outside all previously added spheres, adds the sphere centered at that point whose radius is the distance to the surface. In other words, the largest spheres are added first, and no sphere contains the center

of another sphere (Figure 2-2). Although the procedure described above takes $O(nb)$ time in the worst case (where n is the number of points, and b is the final number of spheres inserted), worst case behavior is rarely seen because most points are processed while there is a small number of large spheres. In fact, this step typically takes less than 1% of the time of the entire algorithm.

Graph Construction The final discretization step constructs the edges of the graph by connecting some pairs of sphere centers (Figure 2-3). Pinocchio adds an edge between two sphere centers if the spheres intersect. We would also like to add edges between spheres that do not intersect if that edge is well inside the surface and if that edge is "essential." For example, the neck and left shoulder spheres of the character in Figure 2-2 are disjoint, but there should still be an edge between them. The precise condition Pinocchio uses is that the distance from any point of the edge to the surface must be at least half of the radius of the smaller sphere, and the closest sphere centers to the midpoint of the edge must be the edge endpoints. The latter condition is equivalent to the requirement that additional edges must be in the Gabriel graph of the sphere centers (see e.g. [31]). While other conditions can be formulated, we found that the Gabriel graph provides a good balance between sparsity and connectedness.

Pinocchio precomputes the shortest paths between all pairs of vertices in this graph to speed up penalty function evaluation.

2.3.2 Reduced Skeleton

The discretization stage constructs a geometric graph $G = (V, E)$ into which Pinocchio needs to embed the given skeleton in an optimal way. The skeleton is given as a rooted tree on s joints. To reduce the degrees of freedom, for the discrete embedding, Pinocchio works with a reduced skeleton, in which all bone chains have been merged (all degree two joints, such as knees, eliminated), as shown in Figure 2-4. The reduced skeleton thus has only r joints. This works because once Pinocchio knows where the endpoints of a bone chain are in V, it can compute the intermediate joints

by taking the shortest path between the endpoints and splitting it in accordance with the proportions of the unreduced skeleton. For the humanoid skeleton we use, for example, $s = 18$, but $r = 7$; without a reduced skeleton, the optimization problem would typically be intractable.

Therefore, the discrete skeleton embedding problem is to find the embedding of the reduced skeleton into G, represented by an r-tuple $\mathbf{v} = (v_1, \ldots, v_r)$ of vertices in V, which minimizes a penalty function $f(\mathbf{v})$ that is designed to penalize differences in the embedded skeleton from the given skeleton.

2.3.3 Discrete Penalty Function

The discrete penalty function has great impact on the generality and quality of the results. A good embedding should have the proportions, bone orientations, and size similar to the given skeleton. The paths representing the bone chains should be disjoint, if possible. Joints of the skeleton may be marked as "feet," in which case they should be close to the bottom of the character. Designing a penalty function that satisfies all of these requirements simultaneously is difficult. Instead we found it easier to design penalties independently and then rely on learning a proper weighting for a global penalty that combines each term.

The Setup We represent the penalty function f as a linear combination of k "basis" penalty functions: $f(\mathbf{v}) = \sum_{i=1}^{k} \gamma_i b_i(\mathbf{v})$. Pinocchio uses $k = 9$ basis penalty functions constructed by hand. They penalize short bones, improper orientation between joints, length differences in bones marked symmetric, bone chains sharing vertices, feet away from the bottom, zero-length bone chains, improper orientation of bones, degree-one joints not embedded at extreme vertices, and joints far along bone-chains but close in the graph (see Appendix A.2 for details). We determine the weights $\Gamma = (\gamma_1, \ldots, \gamma_k)$ semi-automatically via a new maximum margin approach inspired by support vector machines.

Suppose that for a single character, we have several example embeddings, each marked "good" or "bad". The basis penalty functions assign a feature vector $\mathbf{b}(\mathbf{v}) =$

$(b_1(\mathbf{v}), \ldots, b_k(\mathbf{v}))$ to each example embedding $\mathbf{v}$. Let $\mathbf{p}_1, \ldots, \mathbf{p}_m$ be the k-dimensional feature vectors of the good embeddings and let $\mathbf{q}_1, \ldots, \mathbf{q}_n$ be the feature vectors of the bad embeddings.

Maximum Margin To provide context for our approach, we review the relevant ideas from the theory of support vector machines. See Burges [9] for a much more complete tutorial. If our goal were to automatically classify new embeddings into "good" and "bad" ones, we could use a support vector machine to learn a maximum margin linear classifier. In its simplest form, a support vector machine finds the hyperplane that separates the $\mathbf{p}_i$'s from the $\mathbf{q}_i$'s and is as far away from them as possible. More precisely, if Γ is a k-dimensional vector with $\|\Gamma\| = 1$, the classification margin of the best hyperplane normal to Γ is $\frac{1}{2}\left(\min_{i=1}^{n} \Gamma^T\mathbf{q}_i - \max_{i=1}^{m} \Gamma^T\mathbf{p}_i\right)$. Recalling that the total penalty of an embedding $\mathbf{v}$ is $\Gamma^T\mathbf{b}(\mathbf{v})$, we can think of the maximum margin Γ as the one that best distinguishes between the best "bad" embedding and the worst "good" embedding in the training set.

In our case, however, we do not need to classify embeddings, but rather find a Γ such that the embedding with the lowest penalty $f(\mathbf{v}) = \Gamma^T\mathbf{b}(\mathbf{v})$ is likely to be good. To this end, we want Γ to distinguish between the best "bad" embedding and the *best* "good" embedding, as illustrated in Figure 2-5. We therefore wish to maximize the optimization margin (subject to $\|\Gamma\| = 1$), which we define as:

$$\min_{i=1}^{n} \Gamma^T\mathbf{q}_i - \min_{i=1}^{m} \Gamma^T\mathbf{p}_i.$$

Because we have different characters in our training set, and because the embedding quality is not necessarily comparable between different characters, we find the Γ that maximizes the minimum margin over all of the characters.

Our approach is similar to margin-based linear structured classification [60], the problem of learning a classifier that to each problem instance (cf. character) assigns the discrete label (cf. embedding) that minimizes the dot product of a weights vector with basis functions of the problem instance and label. The key difference is that

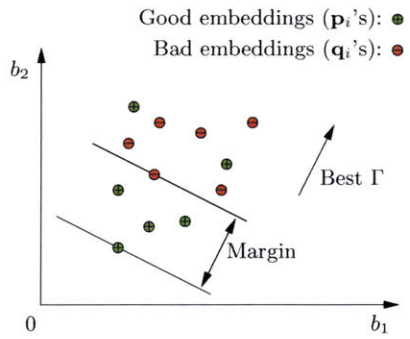

Figure 2-5: Illustration of optimization margin: marked skeleton embeddings in the space of their penalties (b_i's)

structured classification requires an explicit loss function (in our case, the knowledge of the quality of all possible skeleton embeddings for each character in the training set), whereas our approach only makes use of the loss function on the training labels and allows for the possibility of multiple correct labels. This possibility of multiple correct skeleton embeddings prevented us from formulating our margin maximization problem as a convex optimization problem. However, multiple correct skeleton embeddings are necessary for our problem in cases such as the hand joint being embedded into different fingers.

Learning Procedure The problem of finding the optimal Γ does not appear to be convex. However, an approximately optimal Γ is acceptable, and the search space dimension is sufficiently low (9 in our case) that it is feasible to use a continuous optimization method. We use the Nelder-Mead method [48] starting from random Γ's. We start with a cube $[0,1]^k$, pick random normalized Γ's, and run Nelder-Mead from each of them. We then take the best Γ, use a slightly smaller cube around it, and repeat.

To create our training set of embeddings, we pick a training set of characters, manually choose Γ, and use it to construct skeleton embeddings of the characters. For every character with a bad embedding, we manually tweak Γ until a good embedding is produced. We then find the maximum margin Γ as described above and use this new Γ to construct new skeleton embeddings. We manually classify the embeddings

that we have not previously seen, augment our training set with them, and repeat the process. If Γ eventually stops changing, as happened on our training set, we use the found Γ. It is also possible that a positive margin Γ cannot be found, indicating that the chosen basis functions are probably inadequate for finding good embeddings for all characters in the training set.

For training, we used 62 different characters (Cosmic Blobs models, free models from the web, scanned models, and Teddy models), and Γ was stable with about 400 embeddings. The weights we learned resulted in good embeddings for all of the characters in our training set; we could not accomplish this by manually tuning the weights. Examining the optimization results and the extremal embeddings also helped us design better basis penalty functions.

Although this process of finding the weights is labor-intensive, it only needs to be done once. According to our tests, if the basis functions are carefully chosen, the overall penalty function generalizes well to both new characters and new skeletons. Therefore, a novice user will be able to use the system, and more advanced users will be able to design new skeletons without having to learn new weights.

2.3.4 Discrete Embedding

Computing a discrete embedding that minimizes a general penalty function is intractable because there are exponentially many embeddings. However, if it is easy to estimate a good lower bound on f from a partial embedding (of the first few joints), it is possible to use a branch-and-bound method. Pinocchio uses this idea: it maintains a priority queue of partial embeddings ordered by their lower bound estimates. At every step, it takes the best partial embedding from the queue, extends it in all possible ways with the next joint, and pushes the results back on the queue. The first full embedding extracted is guaranteed to be the optimal one. This is essentially the A* algorithm on the tree of possible embeddings. To speed up the process and conserve memory, if a partial embedding has a very high lower bound, it is rejected immediately and not inserted into the queue.

Although this algorithm is still worst-case exponential, it is fast on most real

problems with the skeletons we tested. We considered adapting an approximate graph matching algorithm, like [22], which would work much faster and enable more complicated reduced skeletons. However, computing the exact optimum simplified penalty function design and debugging.

The joints of the skeleton are given in order, which induces an order on the joints of the reduced skeleton. Referring to the joints by their indices (starting with the root at index 1), we define the parent function p_R on the reduced skeleton, such that $p_R(i)$ (for $1 < i \leq r$) is the index of the parent of joint i. We require that the order in which the joints are given respects the parent relationship, i.e. $p_R(i) < i$.

Our penalty function (f) can be expressed as the sum of independent functions of bone chain endpoints (f_i's) and a term (f_D) that incorporates the dependence between different joint positions. The dependence between joints that have not been embedded can be ignored to obtain a lower bound on f. More precisely, f can be written as:

$$f(v_1, \ldots, v_r) = \sum_{i=2}^{r} f_i(v_i, v_{p_R(i)}) + \sum_{i=2}^{r} f_D(v_1, \ldots, v_i).$$

A lower bound when the first k joints are embedded is then:

$$\sum_{i=2}^{k} f_i(v_i, v_{p_R(i)}) + \sum_{i=2}^{k} f_D(v_1, \ldots, v_i) + \\ + \sum_{\{i>k | p_R(i) \leq k\}} \min_{v_i \in V} f_i(v_i, v_{p_R(i)})$$

If f_D is small compared to the f_i's, as is often the case for us, the lower bound is close to the true value of f.

Because of this lower bound estimate, the order in which joints are embedded is very important to the performance of the optimization algorithm. High degree joints should be embedded first because they result in more terms in the rightmost sum of the lower bound, leading to a more accurate lower bound. For example, our biped skeleton has only two joints of degree greater than two, so after Pinocchio has embedded them, the lower bound estimate includes f_i terms for all of the bone chains.

Because there is no perfect penalty function, discrete embedding will occasionally

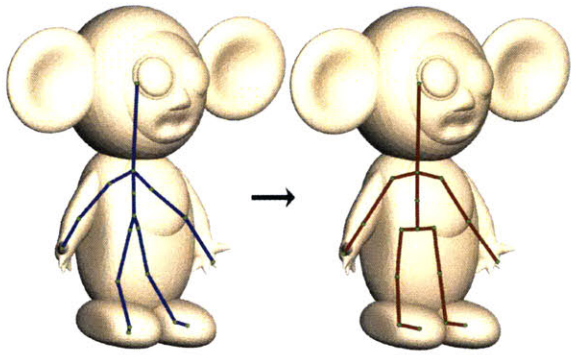

Figure 2-6: The embedded skeleton after discrete embedding (blue) and the results of embedding refinement (dark red)

produce undesirable results (see Model 13 in Figure 2-8). In such cases it is possible for the user to provide manual hints in the form of constraints for reduced skeleton joints. For example, such a hint might be that the left hand of the skeleton should be embedded at a particular vertex in G (or at one of several vertices). Embeddings that do not satisfy the constraints are simply not considered by the algorithm.

2.3.5 Embedding Refinement

Pinocchio takes the optimal embedding of the reduced skeleton found by discrete optimization and reinserts the degree-two joints by splitting the shortest paths in G in proportion to the given skeleton. The resulting skeleton embedding should have the general shape we are looking for, but typically, it will not fit nicely inside the character. Also, smaller bones are likely to be incorrectly oriented because they were not important enough to influence the discrete optimization. Embedding refinement corrects these problems by minimizing a new continuous penalty function (Figure 2-6).

For the continuous optimization, we represent the embedding of the skeleton as an s-tuple of joint positions $(q_1, \ldots, q_s)$ in $\mathbb{R}^3$. Because we are dealing with an unreduced skeleton, and discrete optimization has already found the correct general shape, the penalty function can be much simpler than the discrete penalty function. The continuous penalty function g that Pinocchio tries to minimize is the sum of penalty

functions over the bones plus an asymmetry penalty:

$$g(q_1, \ldots, q_s) = \alpha_A g^A(q_1, \ldots, q_s) + \sum_{i=2}^{s} g_i(q_i, q_{p_S(i)})$$

where p_S is the parent function for the unreduced skeleton (analogous to p_R). Each g_i penalizes bones that do not fit inside the surface nicely, bones that are too short, and bones that are oriented differently from the given skeleton: $g_i = \alpha_S g_i^S + \alpha_L g_i^L + \alpha_O g_i^O$. Unlike the discrete case, we choose the α's by hand because there are only four of them, described in Appendix A.3.

Any continuous optimization technique [19] should produce good results. Pinocchio uses a gradient descent method that takes advantage of the fact that there are relatively few interactions. As a subroutine, it uses a step-doubling line search: starting from a given point (in $\mathbb{R}^{3s}$), it takes steps in the given optimization direction, doubling step length until the penalty function increases. Pinocchio intersperses a line search in the gradient direction with line searches in the gradient direction projected onto individual bones. Repeating the process 10 times is usually sufficient for convergence.

2.4 Skin Attachment

The character and the embedded skeleton are disconnected until skin attachment specifies how to apply deformations of the skeleton to the character mesh. Although we could make use of one of the various mesh editing techniques for the actual mesh deformation, we choose to focus on the standard linear blend skinning (LBS) method because of its widespread use. If $\mathbf{v}_j$ is the position of vertex j, $\mathbf{T}^i$ is the transformation of the i^{th} bone, and w_j^i is the weight of the i^{th} bone for vertex j, LBS gives the position of the transformed vertex j as $\sum_i w_j^i \mathbf{T}^i(\mathbf{v}_j)$. The attachment problem is finding bone weights $\mathbf{w}^i$ for the vertices—how much each bone transform affects each vertex.

There are several properties we desire of the weights. First of all, they should not depend on the mesh resolution. Second, for the results to look good, the weights need

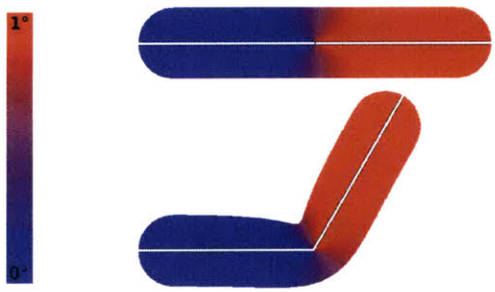

Figure 2-7: Top: heat equilibrium for two bones. Bottom: the result of rotating the right bone with the heat-based attachment

to vary smoothly along the surface. Finally, to avoid folding artifacts, the width of a transition between two bones meeting at a joint should be roughly proportional to the distance from the joint to the surface. Although a scheme that assigns bone weights purely based on proximity to bones can be made to satisfy these properties, such schemes will often fail because they ignore the character's geometry: for example, part of the torso may become attached to an arm. Instead, we use the analogy to heat equilibrium to find the weights. Suppose we treat the character volume as an insulated heat-conducting body and force the temperature of bone i to be 1 while keeping the temperature of all of the other bones at 0. Then we can take the equilibrium temperature at each vertex on the surface as the weight of bone i at that vertex. Figure 2-7 illustrates this in two dimensions.

Solving for heat equilibrium over a volume would require tessellating the volume and would be slow. Therefore, for simplicity, Pinocchio solves for equilibrium over the surface only, but at some vertices, it adds the heat transferred from the nearest bone. The equilibrium over the surface for bone i is given by $\frac{\partial \mathbf{w}^i}{\partial t} = \Delta \mathbf{w}^i + \mathbf{H}(\mathbf{p}^i - \mathbf{w}^i) = 0$, which can be written as

$$-\Delta \mathbf{w}^i + \mathbf{H}\mathbf{w}^i = \mathbf{H}\mathbf{p}^i, \tag{2.1}$$

where Δ is the discrete surface Laplacian, calculated with the cotangent formula [46], $\mathbf{p}^i$ is a vector with $p^i_j = 1$ if the nearest bone to vertex j is i and $p^i_j = 0$ otherwise, and $\mathbf{H}$ is the diagonal matrix with H_{jj} being the heat contribution weight of the nearest bone to vertex j. Because Δ has units of length^{-2}, so must $\mathbf{H}$. Letting $d(j)$

be the distance from vertex j to the nearest bone, Pinocchio uses $H_{jj} = c/d(j)^2$ if the shortest line segment from the vertex to the bone is contained in the character volume and $H_{jj} = 0$ if it is not. It uses the precomputed distance field to determine whether a line segment is entirely contained in the character volume. For $c \approx 0.22$, this method gives weights with similar transitions to those computed by finding the equilibrium over the volume. Pinocchio uses $c = 1$ (corresponding to anisotropic heat diffusion) because the results look more natural. When k bones are equidistant from vertex j, heat contributions from all of them are used: p_j is $1/k$ for all of them, and $H_{jj} = kc/d(j)^2$.

Equation (2.1) is a sparse linear system, and the left hand side matrix $-\Delta + \mathbf{H}$ does not depend on i, the bone we are interested in. Thus we can factor the system once and back-substitute to find the weights for each bone. Because $-\Delta$ can be written as $\mathbf{DS}$, where $\mathbf{D}$ is positive and diagonal and $\mathbf{S}$ is symmetric positive definite, $-\Delta + \mathbf{H} = \mathbf{D}(\mathbf{S} + \mathbf{D}^{-1}\mathbf{H})$, where $\mathbf{S} + \mathbf{D}^{-1}\mathbf{H}$ can be factored by a sparse symmetric positive definite solver. Pinocchio uses the TAUCS [64] library for this computation. Note also that the weights $\mathbf{w}^i$ sum to 1 for each vertex: if we sum (2.1) over i, we get $(-\Delta + \mathbf{H}) \sum_i \mathbf{w}^i = \mathbf{H} \cdot \mathbf{1}$, which yields $\sum_i \mathbf{w}^i = \mathbf{1}$.

It is possible to speed up this method slightly by finding vertices that are unambiguously attached to a single bone and forcing their weight to 1. An earlier variant of our algorithm did this, but the improvement was negligible, and this introduced occasional artifacts.

2.5 Motion Transfer

The configuration of a skeleton can be represented as a translation of the root vertex and the rotations of all of the bones relative to the rest pose. The simplest way of transferring this motion to a character attached to an embedded skeleton is to simply apply these transformations relative to the rest pose of the embedded skeleton (translation components of bone transforms are computed to preserve bone lengths). It is important to use the embedded skeleton, rather than the given skeleton, as

the rest pose: imperfectly embedded bones, especially short ones, such as hips and shoulders, undesirably distort the mesh if they are transformed to align with the given bones.

This naive approach has two main problems. The first problem is that the motion transfer algorithm is oblivious to the geometry of the character, potentially causing physically implausible motion as well as character surface self-intersections. The second problem is that differences between the embedded skeleton and the skeleton on which motion is specified affect the motion of end effectors (e.g. hands and feet), leading, for example, to footskate. We do not attempt to address the first problem, and partially address the second to eliminate footskate on the biped skeleton.

Retargetting motion to new skeleton geometry has been studied in many papers (Gleicher [21] has a survey). Pinocchio uses online motion retargetting [11], a fairly simple method, to constrain the feet and pelvis to positions derived from the original motion. Let M be the skeleton on which the motion is defined. The character is uniformly scaled so that the vertical leg length (from the pelvis to the foot) of the embedded skeleton is the same as that of M. The pelvis position is taken from the motion. The feet positions are also taken from the motion, but are translated in the horizontal plane in the rest coordinate system to compensate for differences in the distance between the feet of M and the embedded skeleton. The feet positions and the pelvis are also translated vertically in the rest coordinate system to compensate for the vertical distance from the foot joint to the bottom of the foot (the pelvis is translated by the smaller of the two foot offsets because we prefer bent legs to overextended legs). Online motion retargetting uses inverse rate control to exploit redundancies in the degrees of freedom of the skeleton to drive a motion that satisfies these constraints.

2.6 Results

We evaluate Pinocchio with respect to the three criteria stated in the introduction: generality, quality, and performance. To ensure an objective evaluation, we use inputs

that were not used during development. To this end, once the development was complete, we tested Pinocchio on 16 biped Cosmic Blobs models that we had not previously tried.

2.6.1 Generality

Figure 2-8 shows our 16 test characters and the skeletons Pinocchio embedded. The skeleton was correctly embedded into 13 of these models (81% success). For Models 7, 10 and 13, a hint for a single joint was sufficient to produce a good embedding.

These tests demonstrate the range of proportions that our method can tolerate: we have a well-proportioned human (Models 1–4, 8), large arms and tiny legs (6; in 10, this causes problems), and large legs and small arms (15; in 13, the small arms cause problems). For other characters we tested, skeletons were almost always correctly embedded into well-proportioned characters whose pose matched the given skeleton. Pinocchio was even able to transfer a biped walk onto a human hand, a cat on its hind legs, and a donut.

The most common issues we ran into on other characters were:

- The thinnest limb into which we may hope to embed a bone has a radius of 2τ. Characters with extremely thin limbs often fail because the the graph we extract is disconnected. Reducing τ, however, hurts performance.

- Degree 2 joints such as knees and elbows are often positioned incorrectly within a limb. We do not know of a reliable way to identify the right locations for them: on some characters they are thicker than the rest of the limb, and on others they are thinner.

Although most of our tests were done with the biped skeleton, we have also used other skeletons for other characters (Figure 2-9).

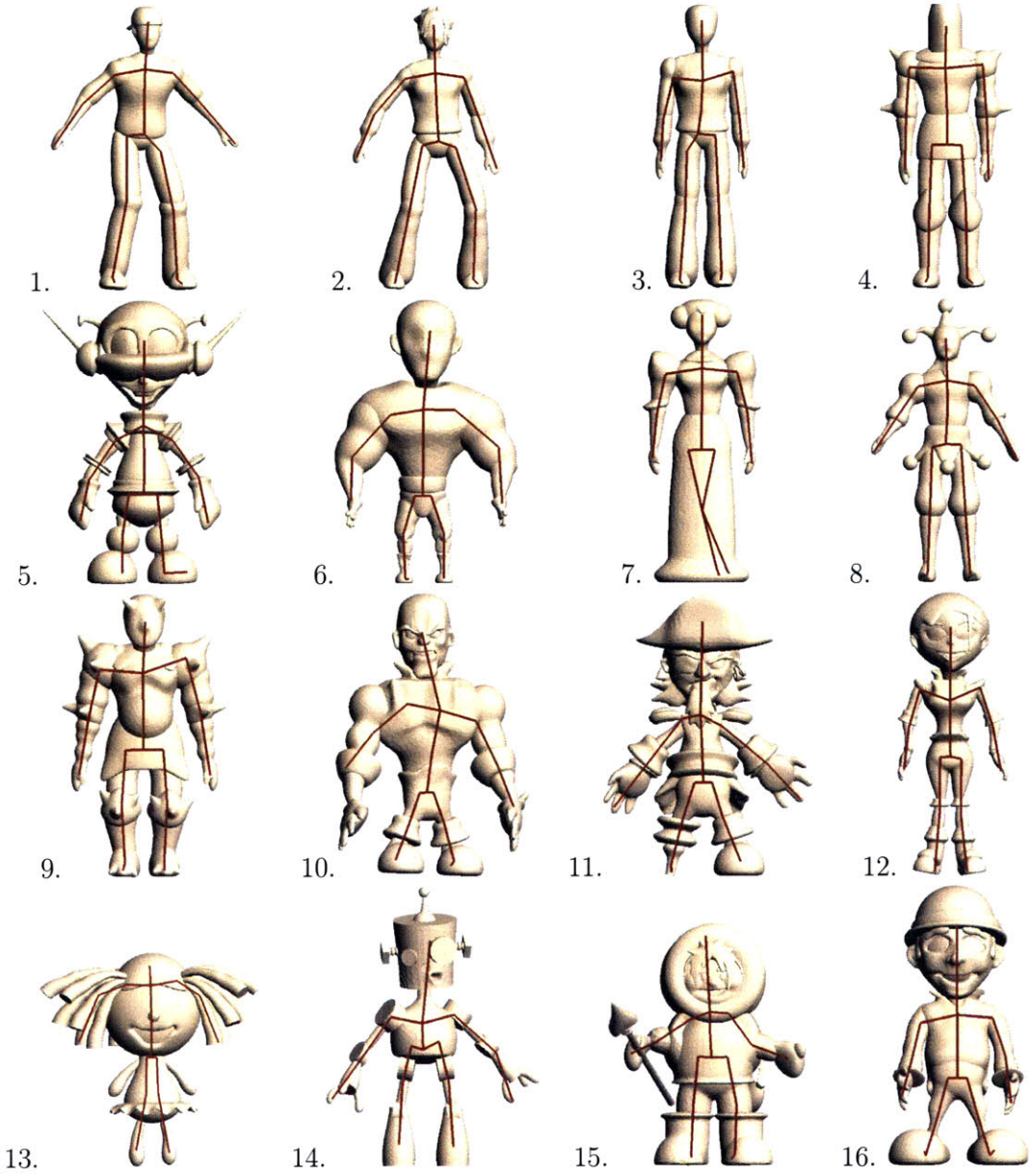

Figure 2-8: Test Results for Skeleton Embedding

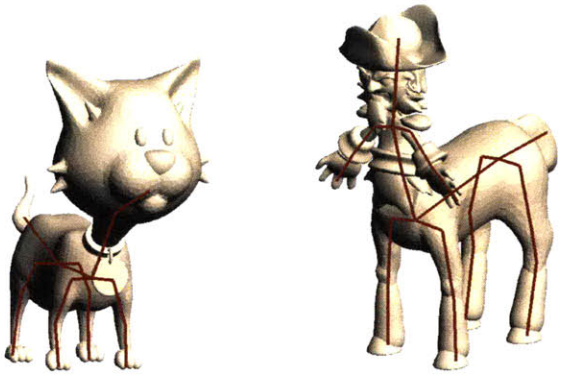

Figure 2-9: A centaur pirate with a centaur skeleton embedded looks at a cat with a quadruped skeleton embedded

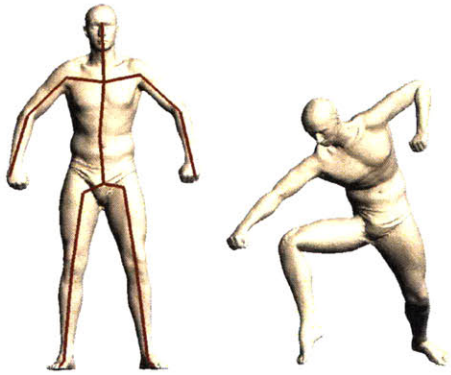

Figure 2-10: The human scan on the left is rigged by Pinocchio and is posed on the right by changing joint angles in the embedded skeleton. The well-known deficiencies of LBS can be seen in the right knee and hip areas.

2.6.2 Quality

Figure 2-10 shows the results of manually posing a human scan using our attachment. Our video [4] demonstrates the quality of the animation produced by Pinocchio.

The quality problems of our attachment are a combination of the deficiencies of our automated weights generation as well as those inherent in LBS. A common class of problems is caused by Pinocchio being oblivious to the material out of which the character is made: the animation of both a dress and a knight's armor has an unrealistic, rubbery quality. Other problems occur at difficult areas, such as hips and the shoulder/neck region, where hand-tuned weights could be made superior to those found by our algorithm.

Model	3	10	11	Mean
Number of Vertices	19,001	34,339	56,856	33,224
Discretization Time	10.3s	25.8s	68.2s	24.3s
Embedding Time	1.4s	29.1s	5.7s	5.2s
Attachment Time	0.9s	1.9s	3.2s	1.8s
Total Time	12.6s	56.8s	77.1s	31.3s

Table 2.1: Timings for three representative models and the mean over our 16 character test set

2.6.3 Performance

Table 2.1 shows the fastest and slowest timings of Pinocchio rigging the 16 models discussed in Section 2.6.1 on a 1.73 MHz Intel Core Duo with 1GB of RAM. Pinocchio is single-threaded so only one core was used. We did not run timing tests on denser models because someone wishing to create real-time animation is likely to keep the triangle count low. Also, because of our volume-based approach, once the distance field has been computed, subsequent discretization and embedding steps do not depend on the given mesh size.

For the majority of models, the running time is dominated by the discretization stage, and that is dominated by computing the distance field. Embedding refinement takes about 1.2 seconds for all of these models, and the discrete optimization consumes the rest of the embedding time.

2.7 Future Work

We have several ideas for improving Pinocchio that we have not tried. Discretization could be improved by packing ellipsoids instead of spheres. Although this is more difficult, we believe it would greatly reduce the size of the graph, resulting in faster and higher quality discrete embeddings. Animation quality can be improved with a technique [68] that corrects LBS errors by using example meshes, which we could synthesize using slower, but more accurate deformation techniques. A more involved approach would be automatically building a tetrahedral mesh around the embedded skeleton and applying the dynamic deformation method of Capell et al. [10]. Com-

bining retargetting with joint limits should eliminate some artifacts in the motion. A better retargetting scheme could be used to make animations more physically plausible and prevent global self-intersections. Finally, it would be nice to eliminate the assumption that the character must have a well-defined interior. That could be accomplished by constructing a cage around the polygon soup character, rigging the cage, and using cage-based deformation to drive the actual character.

Beyond Pinocchio's current capabilities, an interesting problem is dealing with hand animation to give animated characters the ability to grasp objects, type, or speak sign language. The variety of types of hands makes this challenging (see, for example, Models 13, 5, 14, and 11 in Figure 2-8). Automatically rigging characters for facial animation is even more difficult, but a solution requiring a small amount of user assistance may succeed. Combined with a system for motion synthesis [2], this would allow users to begin interacting with their creations.

Chapter 3

Semantic Deformation Transfer

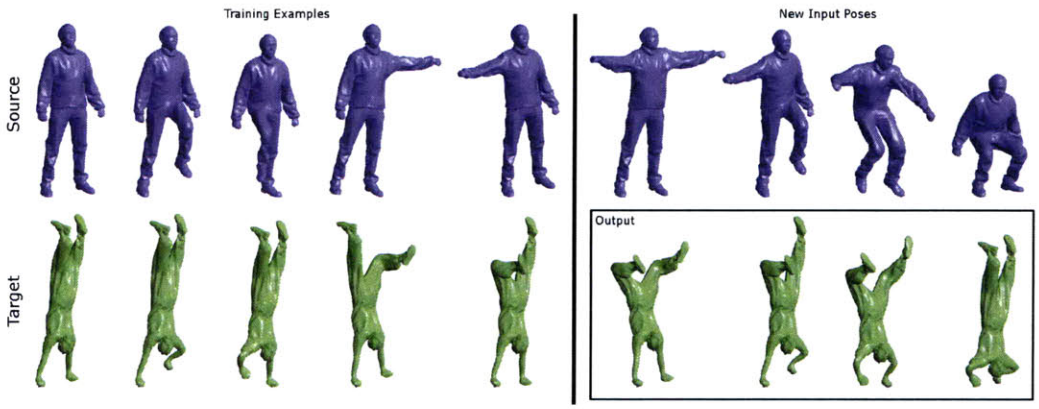

Figure 3-1: Semantic deformation transfer learns a correspondence between poses of two characters from example meshes and synthesizes new poses of the target character from poses of the source character. In this example, given five corresponding poses of two characters (left), our system creates new poses of the bottom character (right) from four poses of the top character.

3.1 Introduction

Advancements in modeling, deformation, and rigging have made the creation of a single character pose a relatively simple task, but creating mesh animations is still time-consuming and laborious. At the same time, recent progress in mesh-based performance capture and deformation transfer has led to an increasing number of

available animations. As a result, reusing mesh animation is emerging as an important problem.

Deformation transfer [58] provides one possible solution. Given a correspondence between two meshes, it copies the deformations of the triangles of the first mesh onto those of the second. The key assumption is that the correspondence is literal: matched parts of the meshes move in geometrically identical ways. Although deformation transfer works well for similar characters and is able to transfer subtle motion details, semantic correspondence is often desirable. The distinction between literal and semantic correspondence can be illustrated with an example of two mesh characters, Alex and Bob. If Alex is walking normally and Bob is walking on his hands, literal correspondence maps Alex's legs to Bob's legs and Alex's arms to Bob's arms, while semantic correspondence maps Alex's legs to Bob's arms and, possibly, vice versa (see Figure 3-1). The ability to transfer motion with semantic correspondence expands the range of potential applications, enabling transfer to drastically different characters that move in unique ways.

Some existing methods could be adapted to transfer motion with semantic correspondence, but with drawbacks. If it is possible to find a geometrically corresponding set of end effectors, their motions can be retargeted [20] and the rest of the mesh could be inferred with MeshIK [59]. Alternatively, if the motions are primarily skeletal, the user can build a skinning model for the target mesh and use a skeletal retargetting method [16, 24]. Both solutions complicate workflow and impose undesirable constraints on the types of transfer that can take place and on the information that the user has to provide. Properly retargetting end effectors requires adjusting the entire time curve, while a skeletal model may not be able to capture the full subtlety of the poses.

Semantic deformation transfer allows the user to specify semantic correspondence (instead of a literal mesh correspondence) by providing examples of corresponding poses of Alex and Bob. To infer the correspondence between two characters and map new poses of one onto the other, semantic deformation transfer represents each pose as a point in a high-dimensional Euclidean "shape space," enabling the use of

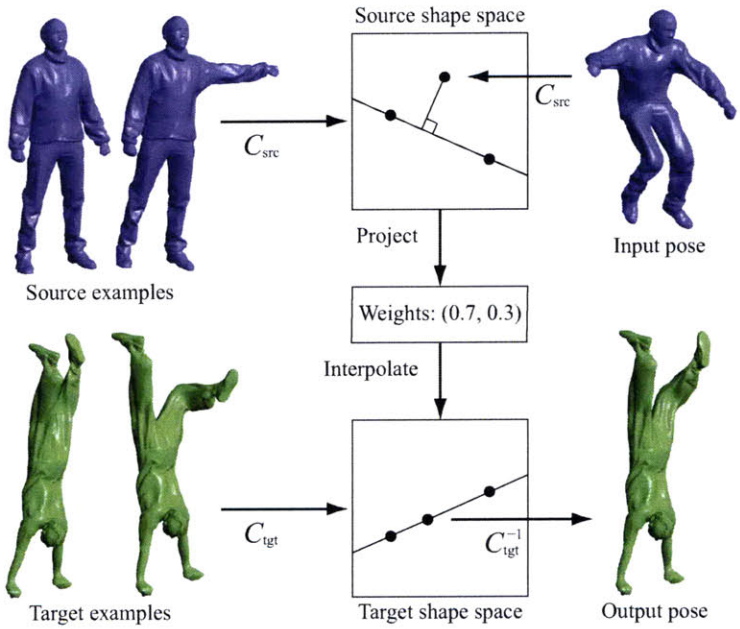

Figure 3-2: Semantic deformation transfer maps the input pose into the source shape space, projects it onto the affine span of example poses, uses the obtained weights to interpolate target example poses in the target shape space, and reconstructs the target pose.

standard linear algebra tools. Using the example poses, semantic deformation transfer constructs a linear map from the source to the target shape space. Given a new source pose, semantic deformation transfer encodes it into the source shape space, maps it to the target shape space, and reconstructs the result to obtain a corresponding target pose (see Figure 3-2).

For semantic deformation transfer to work, the shape space must satisfy two requirements: linear interpolation between points in the shape space must produce blended poses without artifacts, and projection of a pose onto a subspace of the shape space must produce the most similar pose in the subspace to the original. We provide a shape space that meets these requirements.

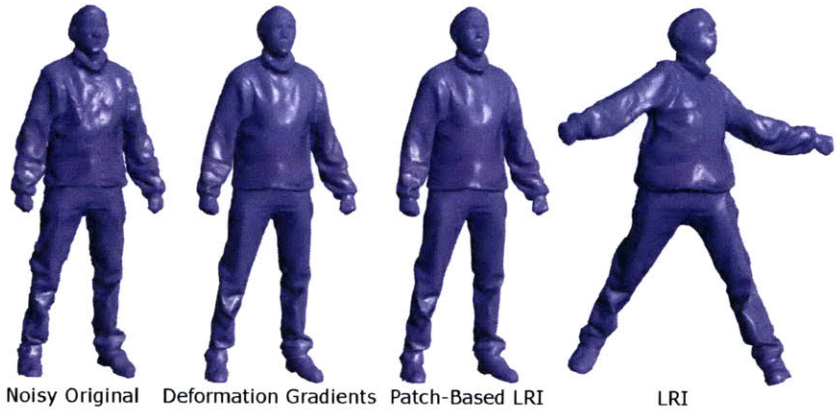

Figure 3-3: The rest pose from Figure 3-1 is corrupted by a small amount of high frequency noise (left). Projecting it to the subspace spanned by the five training poses in Figure 3-1 recovers the pose, if the projection is done in the space of deformation gradient coordinates, or our patch-based LRI coordinates, but not in linear rotation-invariant coordinates. This projection error causes shaking artifacts when transferring from an imperfect source motion using LRI coordinates.

3.2 Shape Space

The proper choice of the shape space is critical for semantic deformation transfer. Several existing mesh representations [59, 43, 39] come close to satisfying this requirement and we combine them into a hybrid representation that enables semantic deformation transfer.

3.2.1 Transfer

A shape space for a particular mesh connectivity is defined by an encoding map C: $\mathbb{R}^{3n} \to \mathbb{R}^m$ that takes mesh vertex positions and outputs a coordinate vector, and by a reconstruction map C^{-1} that returns vertex positions from a coordinate vector. The reconstruction of an encoding must return the original vertex positions (but we do not require multiple encodings to have distinct reconstructions).

Semantic deformation transfer relies on two basic operations in the shape space $\mathbb{R}^m$:

1. *Interpolation:* Given p example poses $\mathbf{x}_1, \ldots, \mathbf{x}_p$ and p weights $w_1, \ldots, w_p$ such

that $\sum_i w_i = 1$, compute $\sum_i w_i \mathbf{x}_i$, the affine combination of the poses.

2. *Projection:* Given p example poses $\mathbf{x}_1, \ldots, \mathbf{x}_p$ and another pose $\mathbf{q}$, compute p weights $w_1, \ldots, w_p$ that minimize

$$\left\| \mathbf{q} - \sum_{i=1}^{p} w_i \mathbf{x}_i \right\| \quad \text{subject to} \quad \sum_i w_i = 1.$$

Letting the cross denote the pseudoinverse, the solution is:

$$\begin{bmatrix} w_2 \\ \vdots \\ w_p \end{bmatrix} = [\mathbf{x}_2 - \mathbf{x}_1 \quad \mathbf{x}_3 - \mathbf{x}_1 \quad \ldots \quad \mathbf{x}_p - \mathbf{x}_1]^\dagger [\mathbf{q} - \mathbf{x}_1],$$

and $w_1 = 1 - \sum_{i=2}^{p} w_i$.

We focus on affine rather than linear interpolation and projection because the origin of the shape space has no special meaning for the transfer.

A shape space is suitable for interpolation if affine combinations of several poses do not result in shrinking or other artifacts (at least when the coefficients are not too negative). Suitability for projection means that the projection of a pose onto an affine span of other poses must retain the characteristics of the unprojected pose as much as possible. For example, random high-frequency noise must be nearly orthogonal to meaningful pose changes (see Figure 3-3) and deformations in different parts of the mesh should be nearly orthogonal to each other (see Figure 3-5).

A shape space that supports interpolation and projection enables semantic deformation transfer with the following simple algorithm (Figure 3-2):

1. Given p pairs of example poses, encode them into the source and target shape spaces using C_{src} and C_{tgt}. Precompute the pseudoinverse (using singular value decomposition) for projection in the source space.

2. Given a new source pose, encode it into the source shape space and use projection to express it as an affine combination of the source example poses with weights $w_1, \ldots, w_p$.

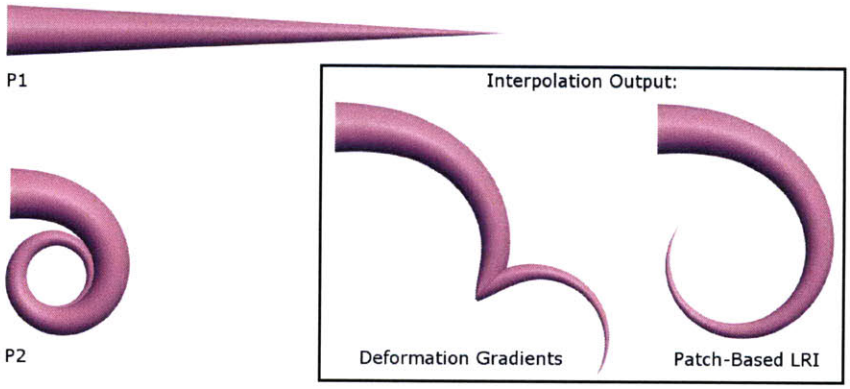

Figure 3-4: Interpolating halfway between two "poses" of this cone, $P1$ and $P2$, fails with deformation gradient coordinates, but works with patch-based LRI coordinates.

3. Use these weights to interpolate the corresponding target example poses in their shape space and use C_{tgt}^{-1} to reconstruct the resulting pose.

Together, the projection and interpolation comprise a linear map from the source shape space to the target shape space.

The above method transfers the aspects of the pose spanned by the example poses. However, global rotation and translation often depend on the pose in a complicated way (e.g. through foot plants or dynamics), and the above method does not take this into account. We therefore ignore the reconstructed global orientation and use heuristics for some of our animations: we apply the average rotation from the source to the target directly and obtain the translation by treating the lowest vertex of the output motion as a foot plant.

3.2.2 Existing Shape Representations

In choosing the shape space, an obvious possibility is to use the vertex positions (C is the identity map). This is known to work poorly for interpolation because linearly blending between rotated parts of the mesh does not interpolate rotation and causes shrinking and other artifacts. The inadequacy of vertex positions has led to the development of many mesh representations [7]. Linear mesh representations (C is a linear map), such as Laplacian coordinates, are also unsuitable for interpolation

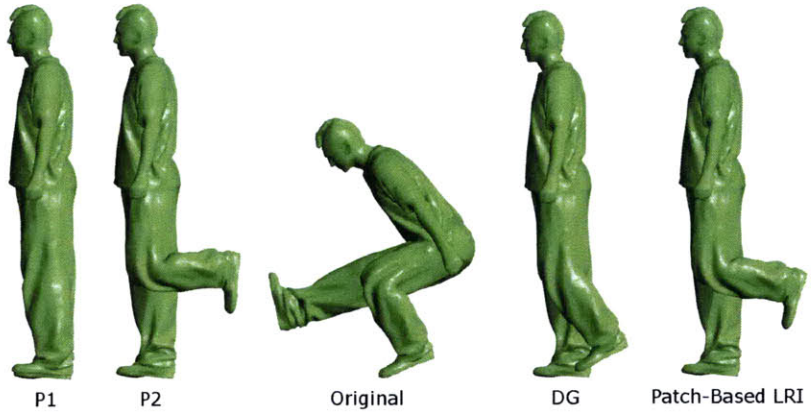

Figure 3-5: The span of the poses $P1$ and $P2$ on the left defines the configuration space of the character's left knee. If we take a pose (middle) and project it onto this subspace, we should recover the knee configuration. The global rotation throws deformation gradient coordinates off (right), while projecting in patch-based LRI coordinates correctly recovers the bent knee.

because they produce the same artifacts as vertex positions: an affine combination in such a space is equivalent to the same affine combination in the vertex position space.

An approach that produces excellent interpolation results is to define a Riemannian metric (instead of the Euclidean metric) on the vertex position space that penalizes non-isometric deformation [38]. However, computation in this space is much more difficult and expensive than in a Euclidean space.

Deformation Gradients One approach to handling rotations is to represent a mesh using deformation gradients to encode individual face transformations. Given two poses, the *deformation gradient* of a mesh face is the matrix that transforms the edge and normal vectors of the face from the first pose to the second. Since translation does not affect the edge and normal vectors, translations are not recorded in the deformation gradient. Let $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v_3}$ be the three vertices of a face and let $\mathbf{n}$ be its scaled normal, computed using

$$\mathbf{n} = (\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{v}_3 - \mathbf{v}_1) / \sqrt{\|(\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{v}_3 - \mathbf{v}_1)\|},$$

following Sumner and Popović [58]. Let $\tilde{\mathbf{v}}_1$, $\tilde{\mathbf{v}}_2$, $\tilde{\mathbf{v}}_3$, and $\tilde{\mathbf{n}}$ be the corresponding vertices and scaled normal in the rest pose. The deformation gradient is the following 3×3 matrix:

$$\mathbf{D} = [\mathbf{v}_2 - \mathbf{v}_1 \quad \mathbf{v}_3 - \mathbf{v}_1 \quad \mathbf{n}] [\tilde{\mathbf{v}}_2 - \tilde{\mathbf{v}}_1 \quad \tilde{\mathbf{v}}_3 - \tilde{\mathbf{v}}_1 \quad \tilde{\mathbf{n}}]^{-1}.$$

A mesh can be represented by recording the deformation gradients of all of the faces relative to a rest pose. For example, MeshIK uses such a representation for projection [59]. However, linearly interpolating deformation gradients does not preserve rotations. Therefore, for interpolation, MeshIK performs a polar decomposition of each deformation gradient $\mathbf{QS} = \mathbf{D}$ and stores $\mathbf{S}$ and $\log \mathbf{Q}$ separately, allowing rotations to be interpolated in logarithmic space.

This representation becomes problematic when the mesh undergoes a large global rotation relative to the rest pose (imagine interpolating the rest pose and a perturbed rest pose rotated 180 degrees: each face rotation would choose a different interpolation path, depending on its perturbation). Factoring out the average deformation gradient rotation (found by using polar decomposition to project $\sum_{f \in \text{faces}} \mathbf{Q}_f$ to a rotation matrix) and storing it separately avoids this problem. We refer to this representation as *deformation gradient coordinates.*

Even with the global rotation factored out, this representation has two more drawbacks. For interpolation, factoring out the average rotation may not be enough and interpolating between two poses in which some faces have rotated more than 180 degrees will result in discontinuity artifacts (Figure 3-4). These types of artifacts can often arise in deformations of tails, snakes, and tentacles, for example. For projection, the deformation gradient coordinates are not locally rotation invariant, resulting in dependency between degrees of freedom that should be independent. Figure 3-5 shows an experiment in which we project a pose with a bent back and a bent knee onto the subspace of poses spanning possible knee configurations. In deformation gradient coordinates, the dependence between the bent back and bent knee results in an incorrect projection.

Rotation-Invariant Coordinates Linear rotation-invariant (LRI) coordinates [43] define a coordinate frame at each mesh vertex and encode that vertex's one-neighborhood in essentially cylindrical coordinates in that frame. Because the coordinate frames themselves are not stored, this representation is rotation-invariant. The mesh is efficiently reconstructed by first finding *connection maps* that encode relationships between frames. A connection map is a rotation matrix that represents a frame in the coordinates of an adjacent frame. Using the connection maps, the reconstruction algorithm solves a large least-squares system to reconstruct the absolute frame orientations, and then solves another least squares system to reconstruct vertex positions. Kircher and Garland's relative blending [39] is similar, but frames are non-orthonormal, defined on mesh faces instead of vertices and the connection maps are stored explicitly, rather than encoded in one-neighborhoods. Pyramid coordinates [56] also store local geometry in a rotation-invariant manner, but the reconstruction is nonlinear and thus more costly. LRI coordinates work very well for interpolation (as they were designed with that purpose in mind) and we use them as a starting point to construct our shape space.

The sole reliance on local orientation relationships makes LRI coordinates noise-sensitive for projection, as shown in Figure 3-3. For semantic deformation transfer, this leads to noticeable shaking artifacts, exaggerating imperfections in the input motion (see the accompanying video). We address this problem by defining frames on mesh patches larger than just one-neighborhoods of vertices. In addition to making LRI robust to noise, using larger patches speeds up reconstruction because a much smaller system needs to be factored for each pose.

3.2.3 Patch-Based LRI Coordinates

To define our patch-based LRI coordinates, we extend LRI by partitioning the mesh faces into several contiguous disjoint patches, factoring out the average rotations of these patches, and using these average rotations as frames. This extension requires some care:

- Extending cylindrical coordinates to larger patches directly does not work because deformations of larger patches are likely to have faces that rotate relative to the patch frame. As with Cartesian coordinates, linearly interpolating between rotated triangles in cylindrical coordinates does not (in general) interpolate the rotation. We therefore encode the local geometry of the larger patches using polar decompositions of deformation gradients.

- LRI reconstructs connection maps between frames from overlapping vertex neighborhoods. Using overlapping patches would make reconstruction more expensive: to solve for the patch frames, we would first need to reconstruct the individual patches from local deformation gradients and then reconstruct the entire model from deformation gradients again (so as not to have seams). Instead, like Kircher and Garland [39], we store the connection maps explicitly, but unlike them, we use orthonormal frames because this avoids global shear artifacts in reconstruction that our experiments revealed (see video).

- We encode rotations with matrix logarithms. Compared to the nonlinear quaternion interpolation, linearly blending matrix logarithms is not rotation-invariant and introduces error. Because this interpolation error is smallest when rotations are small or coaxial, we use deformation gradients relative to a rest pose. For the same reason, unlike LRI, we work with patch frames relative to the rest pose. As a result, when we encode the rest pose, all of our connection maps are the identity. Our experiments confirmed that the results are not sensitive to the choice of the rest pose, as long as it is a reasonable pose for the character.

Encoding Let $\mathbf{D}_f$ be the deformation gradient of mesh face f relative to the rest pose and let $\mathbf{Q}_f\mathbf{S}_f = \mathbf{D}_f$ be the polar decomposition of this deformation gradient. Let $\bar{\mathbf{Q}}$ be the average of all $\mathbf{Q}_f$'s (computed by orthonormalizing $\sum_f \mathbf{Q}_f$ using polar decomposition). Let $P_1, \ldots, P_k$ be the patches and let $p(f)$ be the index of the patch to which face f belongs. Let $\mathbf{G}_1, \ldots, \mathbf{G}_k$ be the average rotations of the deformation

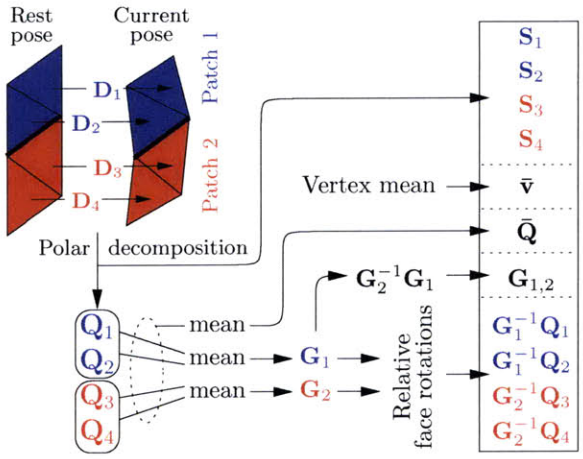

Figure 3-6: A mesh with four faces and two patches is encoded into patch-based LRI coordinates. The rotation matrices are stored as logarithms (i.e. as a vector whose direction is the axis of rotation and whose magnitude is the angle.)

gradients in each patch. We encode into patch-based LRI coordinates by storing the following in a coordinate vector (Figure 3-6):

- the scale/shear components: $\mathbf{S}_f$ for each face,
- the mean vertex position $\bar{\mathbf{v}}$, and the mean face rotation $\log \bar{\mathbf{Q}}$,
- connection maps between patches: $\log(\mathbf{G}_{i,j})$ for each pair (i,j) of adjacent patches, where $\mathbf{G}_{i,j} = (\mathbf{G}_j)^{-1}\mathbf{G}_i$,
- rotations within patches: $\log\left((\mathbf{G}_{p(f)})^{-1}\mathbf{Q}_f\right)$ for each face.

Reconstruction Given such a coordinate vector, we reconstruct the vertex positions using the following algorithm:

1. We first reconstruct each patch's average rotation. To have rotation-invariance, we only store the relative rotations $\mathbf{G}_{i,j}$ between patches, so reconstruction finds $\mathbf{G}_1, \ldots, \mathbf{G}_k$ that minimize

$$\sum_{\{(i,j)|P_i \text{ and } P_j \text{ adjacent}\}} \|\mathbf{G}_i - \mathbf{G}_j\mathbf{G}_{i,j}\|^2.$$

Because the $\mathbf{G}_j$'s are 3-by-3 matrices, this can be converted into a linear least squares system and solved using sparse Cholesky factorization. To make the system well-posed, we select an arbitrary $\mathbf{G}_i$ and constrain it to the identity matrix. Although the system needs to be refactored for every pose, it is small (its size depends only on the number of patches) and this solve is not a bottleneck in the reconstruction. The resulting matrices may not be orthonormal, so at the conclusion of the least-squares solve we use the polar decomposition to project each $\mathbf{G}_i$ to the nearest rotation matrix.

2. Next, we reconstruct the deformation gradient for each mesh face: $\mathbf{D}_f = \mathbf{Q}_f \mathbf{S}_f$. The matrix $\mathbf{S}_f$ is read directly from the coordinate vector and $\mathbf{Q}_f$ is computed by multiplying the average patch rotation $\mathbf{G}_{p(f)}$ found in step 1 by the relative rotation of the face within the patch.

3. The deformation gradients do not give us absolute vertex positions, but applying a deformation gradient to an edge vector of the rest pose gives a desired edge vector for the current pose. To reconstruct the vertex positions $\mathbf{v}'$ (with arbitrary global translation and rotation), we therefore perform a least squares solve, similar to Kircher and Garland [39]. For each face f with corners i_1, i_2, i_3, we find $\mathbf{v}'_1, \ldots, \mathbf{v}'_n$ that minimize

$$\sum_f \sum_{j=1}^{3} (\mathbf{v}'_{i_{j+1}} - \mathbf{v}'_{i_j} - \mathbf{D}_f(\tilde{\mathbf{v}}_{i_{j+1}} - \tilde{\mathbf{v}}_{i_j}))^2,$$

where $\tilde{\mathbf{v}}$ are the rest pose vertex positions and $j+1$ is taken modulo 3. To make this system well-posed, we constrain an arbitrary vertex to the origin. This system can be factored once for a given mesh connectivity using a sparse Cholesky solver and each new pose requires only a back-substitution.

4. We now have a set of vertex positions, but their global position and orientation is arbitrary. We rigidly transform the vertices:

$$\mathbf{v} = \bar{\mathbf{Q}}(\bar{\mathbf{Q}}')^{-1}(\mathbf{v}' - \bar{\mathbf{v}}') + \bar{\mathbf{v}},$$

where $\mathbf{v}'$ is the vertex reconstructed in step 3, $\bar{\mathbf{v}}'$ is the average reconstructed vertex position, $\bar{\mathbf{Q}}'$ is the average reconstructed face orientation, and $\bar{\mathbf{v}}$ and $\bar{\mathbf{Q}}$ are the desired global position and orientation stored in the coordinate vector.

Weights Different elements of the coordinate vector have different scales and we therefore multiply the elements by different weights when encoding (and divide during reconstruction). The relative weights of individual coordinates do not affect interpolation, but need to be chosen properly for the projection to work well. The weight on the global motion is nearly zero because our transfer model does not take global motion into account. The weight of each face rotation within its patch is set to 1. The weight of the relative rotation $\log \mathbf{G}_{i,j}$ is $\sqrt[4]{|P_i||P_j|}$, where $|P_i|$ is the number of faces in patch i (we use the fourth root because the l_2 norm squares the weights). We set a small weight, 0.1, on the scale components because we primarily consider rotation to be a good descriptor of pose. These weights help preserve large-scale shape changes in favor of smaller-scale ones. In principle, we could have individual face weights depending on the face area, but our meshes are relatively uniformly sampled and this has not been necessary.

Partition Two considerations apply when partitioning the mesh into patches. Too many small patches tends to result in shaking artifacts, similar to LRI. On the other hand, a patch that is too large can contain faces rotated by more than 180° relative to the patch frame, leading to artifacts like those for deformation gradient coordinates (see Figure 3-4). A conservative prediction of the range of poses minimizes the risk of these artifacts, although they might still occur. Segmenting a human into five to 50 patches works well in our tests. To partition the mesh into patches, we apply the first stage of the reduced deformable model construction algorithm by Wang and colleagues [68] to our example poses. It starts with each face being a separate patch and merges patches in a bottom-up fashion, while minimizing the error of assuming all faces of a patch deform the same way. We set the error tolerance very high to obtain between five and fifteen patches for each model. Figure 3-7 shows a partition

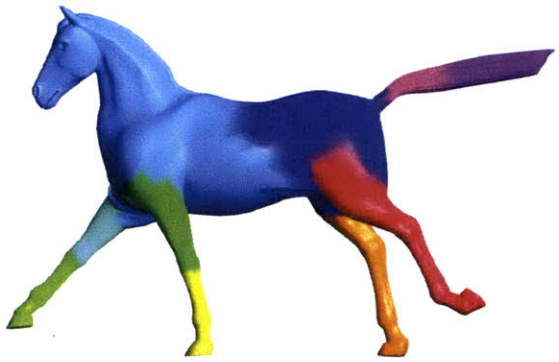

Figure 3-7: For processing the gallop with patch-based LRI coordinates, we split the horse mesh into ten patches.

of one of our test characters.

3.3 Specifying Semantic Correspondence

Combining Existing Poses We assume that the user has some poses of the source and target characters, but not necessarily enough corresponding pose pairs to properly define a semantic correpondence. Our user interface provides a tool to generate new poses, to be used as examples, by combining elements of existing ones. For example, given a pose with a bent knee and a pose with a straight leg, the user can select the knee region and apply the bent knee to the straight leg pose. We accomplish this by transferring the relevant LRI coordinates [43]. The user can select a region of the mesh in one pose, which determines a subset of LRI coordinates (associated with the selected vertices). The user can then apply the shape of that region to another pose. The user can either copy the selected region as-is, or use interpolation/extrapolation to fine-tune its shape.

Extracting Source Poses The key to successful semantic deformation transfer is for the set of example poses of the source character to span the relevant aspects of the motion. We can provide the user with such a set from the motion automatically by finding frames that are farthest from each other. Let V_i be the vertex positions of

the source mesh in $i = 1 \ldots p$ example poses. We start with a rest pose V_1. We set V_2 to be the frame of the motion farthest from V_1, and in general V_i to be the frame farthest from the subspace spanned by V_1 through V_{i-1}. All distances are measured in the shape space of patch-based LRI coordinates. This leaves it to the user to specify only the target's corresponding poses.

Splitting Independent Parts In many cases, the user can reduce the amount of work to construct example poses by decomposing a semantic correspondence into correspondences between independent parts of the meshes. For example, for transfering Alex's normal walk to Bob's walk on his hands, the mapping of Alex's upper body to Bob's lower body can be specified independently from the mapping of Alex's lower body to Bob's upper body. When the user specifies such a decomposition on Alex, our prototype UI extracts Alex's upper body motion separately from his lower body motion (using LRI coordinates from the rest pose to keep the remainder of Alex fixed). It then uses the procedure in the previous paragraph to extract representative poses from both the upper body and the lower body motions. The result is that half of the poses need only the upper body posed and half of the poses only need the lower body posed.

3.4 Results

We applied our method to publicly available mesh animations from performance capture [66] and deformation transfer [58]. The motions we created (available in the accompanying video) are listed in Table 3.1.

Although we did not spend much time optimizing our implementation, it is quite fast. The flamingo is the largest mesh we tested at 52,895 triangles. Encoding a frame of the flamingo into patch-based LRI coordinates takes 0.22 seconds and reconstruction takes 0.25 seconds on a 1.73 Ghz Core Duo laptop. Given the example poses, applying semantic deformation transfer to the 175 frame crane animation takes 136 seconds, including reading the data from disk, partitioning both meshes into

Source motion	Target character	Example poses
Crane	Flamingo	7
Swing	Hand	12
Alex March	Handstand	5
Bob March	Xavier	7
Gallop	Alex and Bob	8 and 6
Squat	Cone	2

Table 3.1: Generated results (the number of example poses includes the rest pose).

patches, building the linear map, applying it, and reconstructing.

3.5 Discussion

With semantic deformation transfer we obtained a variety of useful results in an intuitive manner (Figure 3-8). The ability to treat a pose as a point in Euclidean space enables the use of tools from linear algebra and simplifies processing.

Our simple transfer model, while adequate for many motions, is incapable of representing nonlinear requirements (e.g. the target knee should only bend when the source knee is bent more than 45 degrees). A more sophisticated model, such as radial basis functions, could handle a wider range of transfers. Our shape space makes it possible to explore these directions. Future work should also automatically determine the global motion. This is difficult because the global motion often depends on the pose in a complicated way (e.g., through foot plants or dynamics).

Given the patch-based nature of our shape space representation, one might expect to see seam artifacts between patches. A seam artifact appears when the interpolated rotation of a face differs significantly depending on the patch to which it is assigned. This difference results from the inconsistency between logarithmic blending of connection maps and face rotations relative to the frame. This inconsistency tends to be small and is masked somewhat by the least squares vertex reconstruction step, so the effects are not visible in any of our examples. We constructed synthetic deformations of a cone in which faces rotate along different axes from the rest of the patch and were able to get slight seam artifacts to appear when blending with patch-based LRI

Figure 3-8: A dancer's pose is mapped to a large hand and a man's pose to a flamingo.

coordinates . Should seam artifacts appear in a real motion, they could be eliminated by making the partition into patches "soft."

Chapter 4

Conclusion

We presented two methods that assist an artist trying to create character animation. The first method is for automatically rigging an unfamiliar character for skeletal animation. In conjunction with existing techniques, it allows a user to go from a static mesh to an animated character quickly and effortlessly. We have shown that using this method, Pinocchio can animate a wide range of characters. We also believe that some of our techniques, such as finding LBS weights or using examples to learn the weights of a linear combination of penalty functions, can be useful in other contexts.

The paper describing the automatic rigging method was published in 2007 [3] and the source to Pinocchio released under a permissive license. Since then, our method for finding LBS weights has been implemented in the free modeling package Blender (under the name "Bone Heat") and plugins are available for Maya. The response from artists has been very positive: claims of enhanced productivity abound. A prototype game, called Phorm, based on Pinocchio has also been developed by GAMBIT, the Singapore-MIT game lab. It uses Pinocchio to enable the player to modify his or her character's geometry as part of the gameplay.

Since the original publication, several methods have also proposed alternatives or improvements to our method. To improve LBS weights, a method has been proposed called "Bone Glow" [70]. Bone glow appears to produce better results in some cases, although the paper does not discuss performance. To avoid LBS artifacts and to have more control over the deformation, another recent method almost-automatically con-

structs a cage, based on templates of joint behaviors [34]. The key advantage of both this method and Pinocchio is that the user can specify at once how an entire class of characters is rigged (either with the cage templates, or with the template skeleton). For skeleton embedding, a method based on generalizing Centroidal Voronoi Tessellation has been proposed that makes fewer assumptions about the character than our method [45].

Semantic deformation transfer has been published more recently and we do not yet have the benefit of hindsight. However, some possibilities for further work in this direction are intriguing. Although projection in our shape space produces intuitive results, we do not know whether distance in the shape space is a good measure of pose dissimilarity, or, indeed how to measure pose dissimilarity at all. Formulating such a metric would enable a quantitative comparison of different shape spaces and provide a principled way of choosing weights for our coordinates.

Determining the minimum amount of information necessary to specify a transfer is an interesting conceptual and practical challenge: while a surface correspondence enables literal deformation transfer and a few example poses enable semantic deformation transfer, can we perform semantic *motion* transfer using example motion clips? Different characters move with different rhythms in a way that is difficult to capture with just a mapping between their pose spaces. Building a model that takes time or even physics into account could lead to much higher quality automatically generated animations than what is currently possible.

4.1 Simplifying 3D Content Creation

In this work, we presented two specific tools that can reduce the effort required of artists and animators. The question of how to streamline the 3D content creation process in general remains. To date, production studios have sidestepped this problem by hiring more artists. Nonetheless, in principle, it should be possible for a single person to play the part of the "director," specifying artistic style, characters, setting, and story, and have the work of the rest of the team done by intelligent software. This

is, of course, currently science fiction: I believe we are still at a point when relatively small improvements in tools can have a very large effect on productivity.

There are two obvious directions for improvement: developing new underlying technology that is more flexible and intuitive and improving the user interface. This thesis has almost exclusively focused on the first direction, but without a good interface, technology improvements are useless. In terms of user interfaces, there is a very large gulf between UI research prototypes and production systems.

On the research end, as is inevitable given the limited resources, prototypes are generally quite primitive. While illustrating the authors' idea, these prototypes do not address the scalability concern—whether the idea can function inside a production system. In much otherwise good research (e.g., the Teddy—Fibermesh line of work), the answer seems to be "no."

On the production end, developers have noticed that professional artists are very good at adapting and working around limitations in software, as long as there is a path to their desired end result. This has led to software whose UIs are far more complicated than necessary, requiring many steps for seemingly simple tasks. I have witnessed a professional 3D modeler who was very fast and proficient in Maya spend an hour on a part that could have been modeled in twenty minutes using a recently published system [53].

The unfortunate result of this distance between research and production is that progress has been slow: most modelers, riggers, and animators today don't work much differently than they did ten years ago. Researchers often don't have the capability to evaluate their ideas on a large scale, while deficiencies in production UIs greatly obscure the underlying technical issues that need to be addressed to improve productivity. A clean, extendible, stable toolchain with a well-thought-out UI (as Eclipse is to Java programming, for example) could break the impasse and make life easier for artists (at least those remaining employed) while facilitating research. Building such a system is a very difficult undertaking and given the entrenched market position of existing software, not many entities could attempt it. However, without it, progress will continue to be slow.

Appendix A

Penalty Functions for Automatic Rigging

A.1 Skeleton Joint Attributes

The skeleton joints can be supplied with the following attributes to improve quality and performance (without sacrificing too much generality):

- A joint may be marked symmetric with respect to another joint. This results in symmetry penalties if the distance between the joint and its parent differs from the distance between the symmetric joint and its parent.
- A joint may be marked as a foot. This results in a penalty if the joint is not in the bottommost position.
- A joint may be marked as "fat." This restricts the possible placement of the joint to the center of the σ largest spheres. We use $\sigma = 50$. In our biped skeleton, hips, shoulders and the head are marked as "fat."

A.2 Discrete Penalty Basis Functions

The discrete penalty function measures the quality of a reduced skeleton embedding into the discretized character volume. It is a linear combination of basis penalty

functions (Pinocchio uses nine). The weights were determined automatically by the maximum-margin method described in the paper and are (0.27, 0.23, 0.07, 0.46, 0.14, 0.12, 0.72, 0.05, 0.33) in the order that the penalties are presented.

The specific basis penalty functions were constructed in an ad hoc manner. They are summed over all embedded bones (or joints), as applicable. Slight changes in the specific constants used should not have a significant effect on results. We use the notation $s_{a\to b}^{c\to d}(x)$ to denote the bounded linear interpolation function that is equal to b if $x < a$, d if $x > c$, and $b + (d-b)(x-a)/(c-a)$ otherwise.

Pinocchio uses the following discrete penalty functions:

1. It penalizes short bones: suppose a reduced skeleton bone is embedded at vertices v_1 and v_2, whose spheres have radii r_1 and r_2, respectively. Let d be the shortest distance between v_1 and v_2 in the graph, and let d' be the distance between the joints in the unreduced skeleton. If $d + 0.7(r_1 + r_2) < 0.5d'$, the penalty is infinite. Otherwise, the penalty is

$$\left(S_{0.5\to 0}^{2\to 3}\left(\frac{d'}{d+0.7(r_1+r_2)}\right)\right)^3$$

2. It penalizes embeddings in which directions between embedded joints differ from those in the given skeleton. More precisely, for every pair of joints that are either adjacent or share a common parent in the reduced skeleton, we compute c, the cosine of the angle between the vectors $v_2 - v_1$ and $s_2 - s_1$ where v_1 and v_2 are the joint positions in the embedding and s_1 and s_2 are the joint positions in the given skeleton. The penalty is then infinite if $c < \alpha_1$, and is $0.5\max(0, \alpha_2 \cdot (1-c)^2 - \alpha_3)$ otherwise. If the joints are adjacent in the reduced skeleton, we use $(\alpha_1, \alpha_2, \alpha_3 = (0, 16, 0.1))$ and if they share the parent, we use $(\alpha_1, \alpha_2, \alpha_3 = (-0.5, 4, 0.5))$, a weaker penalty.

3. It penalizes differences in length between bones that are marked as symmetric on the skeleton. Suppose that two bones have been marked symmetric and have been embedded into v_1–v_2 and v_3–v_4 with these vertices having sphere radii r_1,

r_2, r_3, and r_4, respectively. Suppose that the distance along the graph edges between v_1 and v_2 is d_1 and the distance between v_3 and v_4 is d_2. Let

$$q = 0.2 \max\left(\frac{d_1}{d_2}, \frac{d_2}{d_1}\right) + \\ + 0.8 \max\left(\frac{d_1}{d_2 + 0.7(r_3 + r_4)}, \frac{d_2}{d_1 + 0.7(r_1 + r_2)}\right).$$

Then the penalty for this pair of bones is $\max(0, q^3 - 1.2)$.

4. It penalizes two bone chains sharing vertices. If two or more bone chain embedding share a vertex whose distance to the surface is smaller than 0.02, the penalty is infinite. If a bone chain is embedded into a path $v_1, \ldots, v_k$ such that v_1 is the child joint and v_k is the parent joint, and if S is the subset of these joints occupied by a previously embedded bone chain, the penalty is $0.5 + \sum_{v_i \in S} \frac{1}{2i^2}$ if S is not empty.

5. It penalizes joints that are marked as feet if they are not in the bottommost possible position. For each such joint, the penalty is the y coordinate difference between the graph vertex with the minimum y and the vertex into which the joint is embedded.

6. It penalizes bone chains of zero length. This penalty is equal to 1 if a joint and its parent are embedded into the same vertex.

7. It penalizes bone segments that are improperly oriented relative to the given bones. This penalty is calculated for the unreduced skeleton, so we first extract the unreduced embedding, as we do before embedding refinement: we reinsert degree-two joints by splitting the shortest paths in the graph in proportion to the given skeleton. The penalty is then the sum of penalties over each unreduced bone. Let $\vec{u}$ be the vector corresponding to the embedded bone and let $\vec{u}'$ be the vector of the bone in the given skeleton. The penalty per unreduced bone is

$$50\|\vec{u}'\|^2 \left((1-c)s^{0\to 1}_{-0.5\to 6}(c)\right)^2$$

where $c = \frac{\vec{u} \cdot \vec{u}'}{\|\vec{u}\|\|\vec{u}'\|}$.

8. It penalizes degree-one joints that could be embedded farther from their parents and are not. Suppose a degree-one joint is embedded into v_2 and its parent is embedded into v_1 (different from v_2). This penalty is equal to 1 if there is a vertex v_3 adjacent to v_2 in the extracted graph whose sphere is at least 1/2 the radius of the sphere at v_2 and the following two conditions hold:

$$\frac{(v_2 - v_1) \cdot (v_3 - v_1)}{\|v_2 - v_1\|\|v_3 - v_1\|} \geq 0.95$$

and

$$\frac{(v_2 - v_1) \cdot (v_3 - v_2)}{\|v_2 - v_1\|\|v_3 - v_2\|} \geq 0.8.$$

Moreover, to improve optimization performance, we never try embedding a degree-one joint into a vertex v_1 if for every adjacent vertex v_2 there is a vertex v_3 adjacent to v_1 such that the sphere around v_3 is at least 1/2 the radius of the sphere around v_1 and:

$$\frac{(v_3 - v_1) \cdot (v_1 - v_2)}{\|v_3 - v_1\|\|v_1 - v_2\|} \geq 0.8.$$

9. It penalizes joints that are embedded close to each other in the graph, yet are far along bone paths. More precisely, for every pair of joints v_1 and v_2 (that are not adjacent in the reduced skeleton), this penalty is 1 if

$$2d(v_1, v_2) + r_1 + r_2 < d(v_1, v_L) + d(v_2, v_L)$$

where d is the distance along graph edges, r_1 and r_2 are the radii of spheres into whose centers v_1 and v_2 are embedded, and v_L is the embedding of the least common ancestor (in the oriented reduced skeleton) of the two embedded joints.

A.3 Embedding Refinement Penalty Function

This penalty function is used to refine the discrete embedding. It was also constructed ad hoc. It is the weighted sum of the following four penalty functions over all bones. The weights we use are $(\alpha_S, \alpha_L, \alpha_O, \alpha_A) = (15000, 0.25, 2, 1)$ for the respective penalties.

1. Pinocchio penalizes bones that are not near the center of the object. The penalty is the average of

$$r(0.003, \min(m(q_i), 0.001 + \max(0, 0.05 + s(q_i))))$$

over 10 samples q_i on the bone, where $r(a, x)$ is 0 if $x < a$ and is x^2 otherwise, $m(p)$ is the distance from p to the nearest sampled medial surface point, and $s(p)$ is the signed distance from p to the object surface (positive when p is outside).

2. It penalizes bones that are too short when projected onto their counterparts in the given skeleton. Suppose a bone has endpoints q_1 and q_2 in the embedding and endpoints s_1 and s_2 in the given skeleton. The penalty is:

$$\max\left(0.5, \frac{\|s_2 - s_1\|^2}{((q_2 - q_1) \cdot (s_2 - s_1))^2 / \|s_2 - s_1\|^2}\right).$$

3. It penalizes improperly oriented bones. Suppose a bone has endpoints q_1 and q_2 in the embedding and endpoints s_1 and s_2 in the given skeleton. Let θ be the angle between the vectors $q_2 - q_1$ and $s_2 - s_1$. The penalty is $(0.3 + 0.5\theta)^3$ if θ is positive and $10 \cdot (0.3 + 0.5\theta)^3$ if θ is negative.

4. It penalizes asymmetry in bones that are marked symmetric. If a bone has endpoints q_1 and q_2 and its symmetric bone has endpoints q_3 and q_4 then the penalty is:

$$\max\left(1.05, \frac{\|q_1 - q_2\|^2}{\|q_3 - q_4\|^2}, \frac{\|q_3 - q_4\|^2}{\|q_1 - q_2\|^2}\right).$$

This penalty appears in the sum once for every pair of symmetric bones.

Bibliography

[1] David Anderson, James L. Frankel, Joe Marks, Aseem Agarwala, Paul Beardsley, Jessica Hodgins, Darren Leigh, Kathy Ryall, Eddie Sullivan, and Jonathan S. Yedidia. Tangible interaction + graphical interpretation: a new approach to 3d modeling. In *Proceedings of ACM SIGGRAPH 2000*, Annual Conference Series, pages 393–402, July 2000.

[2] Okan Arikan, David A. Forsyth, and James F. O'Brien. Motion synthesis from annotations. *ACM Transactions on Graphics*, 22(3):402–408, July 2003.

[3] Ilya Baran and Jovan Popović. Automatic rigging and animation of 3d characters. *ACM Transactions on Graphics*, 26(3), 2007. In press.

[4] Ilya Baran and Jovan Popović. Pinocchio results video. http://people.csail.mit.edu/ibaran/pinocchio.avi, 2007.

[5] Jules Bloomenthal and Chek Lim. Skeletal methods of shape manipulation. In *Proceedings of the International Conference on Shape Modeling and Applications*, page 44, 1999.

[6] Gunilla Borgefors, Ingela Nystrm, and Gabriella Sanniti Di Baja. Computing skeletons in three dimensions. *Pattern Recognition*, 32(7):1225 – 1236, 1999.

[7] Mario Botsch and Olga Sorkine. On linear variational surface deformation methods. *IEEE Transactions on Visualization and Computer Graphics*, 14(1):213–230, 2008.

[8] Matthew Brand and Aaron Hertzmann. Style machines. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 183–192, July 2000.

[9] C.J.C. Burges. A Tutorial on Support Vector Machines for Pattern Recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

[10] Steve Capell, Seth Green, Brian Curless, Tom Duchamp, and Zoran Popović. Interactive skeleton-driven dynamic deformation. *ACM Transactions on Graphics*, 21(3):586–593, August 2002.

[11] Kwang-Jin Choi and Hyeong-Seok Ko. Online motion retargetting. *Journal of Visualization and Computer Animation*, 11(5):223–235, December 2000.

[12] Nicu D. Cornea, Deborah Silver, and Patrick Min. Curve-skeleton properties, applications, and algorithms. *IEEE Transactions on Visualization and Computer Graphics*, 13(3):530–548, 2007.

[13] Marco da Silva, Yeuhi Abe, and Jovan Popović. Interactive simulation of stylized human locomotion. *ACM Transactions on Graphics*, 27(3):82:1–82:10, August 2008.

[14] Edilson de Aguiar, Carsten Stoll, Christian Theobalt, Naveed Ahmed, Hans-Peter Seidel, and Sebastian Thrun. Performance capture from sparse multi-view video. *ACM Transactions on Graphics*, 27(3):98:1–98:10, August 2008.

[15] Edilson de Aguiar, Christian Theobalt, Marcus Magnor, and Hans-Peter Seidel. Reconstructing human shape and motion from multi-view video. In *Conference on Visual Media Production*, pages 44–51, December 2005.

[16] Mira Dontcheva, Gary Yngve, and Zoran Popović. Layered acting for character animation. *ACM Transactions on Graphics*, 22(3):409–416, July 2003.

[17] Michael S. Floater. Mean value coordinates. *Computer Aided Geometric Design*, 20(1):19–27, March 2003.

[18] Sarah F. Frisken, Ronald N. Perry, Alyn P. Rockwood, and Thouis R. Jones. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of ACM SIGGRAPH 2000*, Annual Conference Series, pages 249–254, July 2000.

[19] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization.* Academic Press, London, 1989.

[20] Michael Gleicher. Retargetting motion to new characters. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 33–42, July 1998.

[21] Michael Gleicher. Comparing constraint-based motion editing methods. *Graphical Models*, 63(2):107–134, March 2001.

[22] Steven Gold and Anand Rangarajan. A graduated assignment algorithm for graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(4):377–388, 1996.

[23] Radek Grzeszczuk, Demetri Terzopoulos, and Geoffrey Hinton. Neuroanimator: Fast neural network emulation and control of physics-based models. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 9–20, July 1998.

[24] Eugene Hsu, Sommer Gentry, and Jovan Popović. Example-based control of human motion. In *2004 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 69–77, July 2004.

[25] Eugene Hsu, Kari Pulli, and Jovan Popović. Style translation for human motion. *ACM Transactions on Graphics*, 24(3):1082–1089, August 2005.

[26] Takeo Igarashi and John F. Hughes. Smooth meshes for sketch-based freeform modeling. In *2003 ACM Symposium on Interactive 3D Graphics*, pages 139–142, April 2003.

[27] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: A sketching interface for 3d freeform design. In *Proceedings of ACM SIGGRAPH 1999*, Annual Conference Series, pages 409–416, July 1999.

[28] Takeo Igarashi, Tomer Moscovich, and John F. Hughes. As-rigid-as-possible shape manipulation. *ACM Transactions on Graphics*, 24(3):1134–1141, August 2005.

[29] Takeo Igarashi, Tomer Moscovich, and John F. Hughes. Spatial keyframing for performance-driven animation. In *Symposium on Computer Animation (SCA)*, pages 107–115, jul 2005.

[30] Doug L. James and Christopher D. Twigg. Skinning mesh animations. *ACM Transactions on Graphics*, 24(3):399–407, August 2005.

[31] Jerzy W. Jaromczyk and Godfried T. Toussaint. Relative neighborhood graphs and their relatives. *Proceedings of IEEE*, 80(9):1502–1517, September 1992.

[32] Pushkar Joshi, Mark Meyer, Tony DeRose, Brian Green, and Tom Sanocki. Harmonic coordinates for character articulation. *ACM Transactions on Graphics*, 26(3):71:1–71:9, July 2007.

[33] Tao Ju, Scott Schaefer, and Joe Warren. Mean value coordinates for closed triangular meshes. *ACM Transactions on Graphics*, 24(3):561–566, August 2005.

[34] Tao Ju, Qian-Yi Zhou, Michiel van de Panne, Daniel Cohen-Or, and Ulrich Neumann. Reusable skinning templates using cage-based deformations. *ACM Transactions on Graphics*, 27(5):122:1–122:10, December 2008.

[35] Sagi Katz and Ayellet Tal. Hierarchical mesh decomposition using fuzzy clustering and cuts. *ACM Transactions on Graphics*, 22(3):954–961, August 2003.

[36] L. Kavan, S. Collins, J. Žára, and C. O'Sullivan. Skinning with dual quaternions. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, page 46. ACM, 2007.

[37] L. Kavan and J. Žára. Spherical blend skinning: a real-time deformation of articulated models. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 9–16. ACM New York, NY, USA, 2005.

[38] Martin Kilian, Niloy J. Mitra, and Helmut Pottmann. Geometric modeling in shape space. *ACM Transactions on Graphics*, 26(3):64:1–64:8, July 2007.

[39] Scott Kircher and Michael Garland. Free-form motion processing. *ACM Transactions on Graphics*, 27(2):12:1–12:13, April 2008.

[40] Paul G. Kry, Doug L. James, and Dinesh K. Pai. EigenSkin: Real time large deformation character skinning in hardware. In *Symposium on Computer Animation (SCA)*, pages 153–160, July 2002.

[41] J. P. Lewis, Matt Cordner, and Nickson Fong. Pose space deformations: A unified approach to shape interpolation and skeleton-driven deformation. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 165–172, July 2000.

[42] Yaron Lipman, Johannes Kopf, Daniel Cohen-Or, and David Levin. Gpu-assisted positive mean value coordinates for mesh deformations. In *Fifth Eurographics Symposium on Geometry Processing*, pages 117–124, July 2007.

[43] Yaron Lipman, Olga Sorkine, David Levin, and Daniel Cohen-Or. Linear rotation-invariant coordinates for meshes. *ACM Transactions on Graphics*, 24(3):479–487, August 2005.

[44] Pin-Chou Liu, Fu-Che Wu, Wan-Chun Ma, Rung-Huei Liang, and Ming Ouhyoung. Automatic animation skeleton using repulsive force field. In *11th Pacific Conference on Computer Graphics and Applications*, pages 409–413, October 2003.

[45] Lin Lu, Bruno Lvy, and Wenping Wang. Centroidal voronoi tessellations for line segments and graphs. Technical report, INRIA - ALICE Project Team, 2009. Accepted pending revision.

[46] Mark Meyer, Mathieu Desbrun, Peter Schröder, and Alan H. Barr. Discrete differential-geometry operators for triangulated 2-manifolds. In *Visualization and Mathematics III*, pages 35–57. Springer-Verlag, Heidelberg, 2003.

[47] L. Moccozet, F. Dellas, N. Magnenat-Thalmann, S. Biasotti, M. Mortara, B. Falcidieno, P. Min, and R. Veltkamp. Animatable human body model reconstruction from 3d scan data using templates. In *CapTech Workshop on Modelling and Motion Capture Techniques for Virtual Environments*, pages 73–79, Zermatt, Switzerland, 2004.

[48] J. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[49] J. Thomas Ngo and Joe Marks. Spacetime constraints revisited. In *Proceedings of SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, pages 343–350, August 1993.

[50] Tom Ngo, Doug Cutrell, Jenny Dana, Bruce Donald, Lorie Loeb, and Shunhui Zhu. Accessible animation and customizable graphics via simplicial configuration

modeling. In *Proceedings of ACM SIGGRAPH 2000*, Annual Conference Series, pages 403–410, July 2000.

[51] Chris Pudney. Distance-ordered homotopic thinning: a skeletonization algorithm for 3d digital images. *Comput. Vis. Image Underst.*, 72(3):404–413, 1998.

[52] Marc H. Raibert and Jessica K. Hodgins. Animation of dynamic legged locomotion. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, pages 349–358, July 1991.

[53] Alec Rivers, Frédo Durand, and Takeo Igarashi. 3d modeling with silhouettes. *ACM Transactions on Graphics*, 29(3), July 2010.

[54] Ryan Schmidt and Karan Singh. Sketch-based procedural surface modeling and compositing using surface trees. *Computer Graphics Forum*, 27(2):321–330, April 2008.

[55] Ryan Schmidt, B. Wyvill, M. C. Sousa, and J. A. Jorge. Shapeshop: Sketch-based solid modeling with blobtrees. In *Eurographics Workshop on Sketch-Based Interfaces and Modeling*, pages 53–62, August 2005.

[56] A. Sheffer and V. Kraevoy. Pyramid coordinates for morphing and deformation. In *Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium on (3DPVT'04)*, pages 68–75, 2004.

[57] J. Starck and A. Hilton. Surface capture for performance based animation. *IEEE Computer Graphics and Applications*, 27(3):21–31, 2007.

[58] Robert W. Sumner and Jovan Popović. Deformation transfer for triangle meshes. *ACM Transactions on Graphics*, 23(3):399–405, August 2004.

[59] Robert W. Sumner, Matthias Zwicker, Craig Gotsman, and Jovan Popović. Mesh-based inverse kinematics. *ACM Transactions on Graphics*, 24(3):488–495, August 2005.

[60] B. Taskar, C. Guestrin, and D. Koller. Max-margin markov networks. In *Advances in Neural Information Processing Systems (NIPS 2003)*, Vancouver, Canada, 2003.

[61] Marek Teichmann and Seth Teller. Assisted articulation of closed polygonal models. In *Computer Animation and Simulation '98*, pages 87–102, August 1998.

[62] Matthew Thorne, David Burke, and Michiel van de Panne. Motion doodles: an interface for sketching character motion. *ACM Transactions on Graphics*, 23(3):424–431, August 2004.

[63] J. Tierny, J.P. Vandeborre, and M. Daoudi. 3d mesh skeleton extraction using topological and geometrical analyses. In *Pacific Graphics*, pages 85–94, 2006.

[64] Sivan Toledo. TAUCS: A library of sparse linear solvers, version 2.2. http://www.tau.ac.il/~stoledo/taucs, 2003.

[65] M. Van De Panne. *Control techniques for physically-based animation.* PhD thesis, University of Toronto Toronto, 1994.

[66] Daniel Vlasic, Ilya Baran, Wojciech Matusik, and Jovan Popović. Articulated mesh animation from multi-view silhouettes. *ACM Transactions on Graphics*, 27(3):97:1–97:9, August 2008.

[67] Lawson Wade. *Automated generation of control skeletons for use in animation.* PhD thesis, The Ohio State University, 2000.

[68] Robert Y. Wang, Kari Pulli, and Jovan Popović. Real-time enveloping with rotational regression. *ACM Transactions on Graphics*, 26(3):73:1–73:9, July 2007.

[69] Xiaohuan Corina Wang and Cary Phillips. Multi-weight enveloping: Least-squares approximation techniques for skin animation. In *ACM SIGGRAPH Symposium on Computer Animation*, pages 129–138, July 2002.

[70] R.J. Wareham and J. Lasenby. Bone glow: an improved method for the assignment of weights for mesh deformation. *Lecture Notes in Computer Science*, 5098:63–71, 2008.

[71] William Welch and Andrew Witkin. Variational surface modeling. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, pages 157–166, July 1992.

[72] William Welch and Andrew Witkin. Free-form shape design using triangulated surfaces. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 247–256, July 1994.

[73] Andrew Witkin and Michael Kass. Spacetime constraints. In *Computer Graphics (Proceedings of SIGGRAPH 88)*, pages 159–168, August 1988.

[74] Kangkang Yin, Kevin Loken, and Michiel van de Panne. Simbicon: Simple biped locomotion control. *ACM Transactions on Graphics*, 26(3):105:1–105:10, July 2007.

[75] Yizhou Yu, Kun Zhou, Dong Xu, Xiaohan Shi, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Mesh editing with poisson-based gradient field manipulation. *ACM Transactions on Graphics*, 23(3):644–651, August 2004.

[76] Kun Zhou, Jin Huang, John Snyder, Xinguo Liu, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Large mesh deformation using the volumetric graph laplacian. *ACM Transactions on Graphics*, 24(3):496–503, August 2005.